Universidad de las Ciencias Informáticas

Facultad 3

Título: Modelo de componentes para el marco de trabajo Sauxe

Trabajo de Diploma para optar por el título de

Ingeniero en Ciencias Informáticas

Autor: Abraham Calás Torres

Tutor(es): Ing. Damián Pérez Alfonso

Ing. Claudia Bravo Batista

La Habana, Junio 2014

DECLARACIÓN DE AUTORÍA

Declaro ser autor de la presente tesis y reconozco a la Universidad de las Ciencias Informáticas los derechos patrimoniales de la misma, con carácter exclusivo.

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

Abraham Calás Torres

_____

Firma del Autor

Ing. Damián Pérez Alfonso

_____

Firma del Tutor

Ing. Claudia Bravo Batista

_____

Firma del Tutor

“Cuando todos piensan igual, nadie está pensando.”

AGRADECIMIENTOS

Este trabajo no fue solo resultado de mi esfuerzo, sino de muchas personas que contribuyeron directa e indirectamente en el mismo.

Quiero agradecer:

A mi tutor Damián por ser mi guía en el mundo de la investigación.

A mi tutora Claudia por su ayuda y compromiso.

A Rene por su confianza en mí.

A Nemury por despertar en mí el bichito de la investigación, muchos de mis logros son gracias su primer impulso.

A Oiner por ser mi gurú del marco de trabajo Sauxe.

A mi novia Danay por amarme y soportarme durante toda la carrera.

A mi mamá por darme todos los medios para estudiar sin complicaciones.

A mi papá por estar orgulloso de todos mis logros.

A todos mis amigos y compañeros que de algún modo influyeron en esta etapa de mi vida.

DEDICATORIA

Dedico este trabajo a mi mamá, mi papá, mi novia, mi hermano y en especial a mi querido Abuelo.

RESUMEN

El Departamento de Tecnología del Centro de Informatización de la Gestión de Entidades perteneciente a la Universidad de las Ciencias Informáticas, centra su labor en el desarrollo del marco de trabajo Sauxe. Este marco de trabajo es una base tecnológica para la construcción de aplicaciones web de gestión, que provee aspectos como la seguridad, la posibilidad de uso para múltiples entidades y el soporte para varios idiomas. Si bien, uno de los objetivos de Sauxe es permitir la reutilización de los componentes que se desarrollan sobre su base, la práctica ha demostrado que estos no son extensibles o personalizables más allá de los sistemas para los que fueron concebidos. La ausencia de un estándar que especifique cómo definir, documentar, empaquetar o integrar componentes, es una de las causas que inciden en el bajo índice de reutilización de los mismos. En esta investigación se propone el diseño de un modelo de componentes que permita estandarizar los procesos relacionados con el desarrollo de componentes sobre Sauxe. Como parte de los objetivos también se describe la implementación del modelo de componentes en el marco de trabajo Sauxe, con el propósito de demostrar la aplicabilidad del mismo. Además, a través de una métrica se presentan los resultados que acreditan que los componentes que se desarrollen según el estándar tienen un factor de reutilización recomendable.

PALABRAS CLAVE: componentes, modelo de componentes, reutilización.

ÍNDICE

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

INTRODUCCIÓN

En las últimas décadas ha ocurrido un aumento en el uso de sistemas informáticos en negocios, industrias e investigaciones. Esta expansión ha propiciado que las empresas que desarrollan programas informáticos se encuentren en una invariable competencia. La demanda a estas empresas está dada no solo por las funcionalidades de los sistemas sino también por características como: usabilidad, robustez, confiabilidad, flexibilidad, adaptabilidad y facilidad de instalación y despliegue (Crnkovik, y otros, 2002). Para satisfacer estas exigencias, los sistemas se vuelven cada vez más complejos y grandes, lo que puede incidir en problemas como: la incorrecta estimación de costos y tiempo, la inadecuada productividad de los desarrolladores, la poca calidad de los sistemas desarrollados y la complejidad en el mantenimiento.

La reutilización puede ser clave para resolver estos tipos de problemas ya que atendiendo a (Sodhi, y otros, 1999) permite implementar o actualizar sistemas haciendo uso de activos de software existentes, previamente revisados y certificados. La efectividad de la reutilización depende del enfoque de desarrollo de software utilizado, específicamente en el Desarrollo de Software Basado en Componentes (DSBC) esta característica es fundamental. Construir y almacenar componentes reusables para su posterior uso es visto como la forma más apropiada de incrementar la producción de software confiable (Lucena, 2002).

Con el objetivo de crear una infraestructura reutilizable que provea aspectos como seguridad, auditoría, interoperabilidad, concurrencia, administración de transacciones y soporte para varios idiomas, el Departamento de Tecnología del Centro de Informatización de la Gestión de Entidades (CEIGE) perteneciente a la Universidad de las Ciencias Informáticas (UCI), desarrolló el marco de trabajo Sauxe. Este marco de trabajo actúa como base tecnológica para la construcción de aplicaciones web de gestión y centra su desarrollo en los requerimientos funcionales, las interfaces de usuario y la lógica del negocio. El uso de Sauxe permite disminuir el tiempo, el esfuerzo y los costos en el desarrollo de sistemas de alta y mediana complejidad. Actualmente existen más de 40 entidades y proyectos del país que lo utilizan (Baryolo, 2012).

Si bien Sauxe pretende seguir el paradigma de desarrollo orientado a componentes para garantizar la reusabilidad, los resultados obtenidos en la investigación de Laura Elena Magón Rodríguez (Rodríguez, 2012) confirman que los componentes de mayor importancia arquitectónicamente no son extensibles o personalizables más allá de sus propósitos.

Por otra parte, las siguientes limitantes también provocan dificultades para crear componentes reutilizables:

- Inexistencia de reglas y estándares que especifiquen cómo definir componentes sobre Sauxe.
- El mecanismo de integración que se utiliza actualmente solo permite dos niveles de integración.
- No está descrito como documentar y gestionar el versionado de los activos de software.
- Los desarrolladores no utilizan el estándar de código definido en el libro de ayuda de Sauxe.

Estas limitaciones dificultan el mantenimiento del sistema y la creación de la documentación correspondiente. También se obstaculiza la reutilización de los componentes en otros entornos así como el control de versiones.

A partir de esta problemática se define como ***problema a resolver***: ¿Cómo aumentar la baja reutilización de los componentes que se desarrollan sobre el marco de trabajo Sauxe?

Queda entonces definido como ***objeto de estudio*** la arquitectura basada en componentes siendo los modelos de componentes el ***campo de acción***.

El ***objetivo general*** de esta investigación es diseñar un modelo de componentes que permita aumentar la reutilización de los componentes desarrollados sobre el marco de trabajo Sauxe, a partir de este objetivo se plantean los ***objetivos específicos*** siguientes:

- Confeccionar el marco teórico de la investigación a partir de la búsqueda y revisión bibliográfica sobre modelos de componentes.
- Definir las pautas para la creación de un modelo de componentes.
- Construir un modelo de componentes para el marco de trabajo Sauxe a partir de las pautas definidas.
- Extender el marco de trabajo Sauxe para que soporte los componentes que cumplan con el modelo definido.
- Validar la aplicabilidad del modelo con la implementación en el marco de trabajo Sauxe.
- Validar la mejora en la reutilización de componentes a través de una métrica para calcular el factor de reutilización.

El autor considera que si se logra estandarizar los componentes desarrollados sobre el marco de trabajo Sauxe a través de un modelo de componentes aumentará la reutilización de los mismos, elemento este que se define en el presente trabajo a razón de ***idea a defender***.

En esta investigación se utilizan los siguientes métodos teóricos y empíricos.

Métodos teóricos: se utiliza el método **histórico-lógico** para realizar un estudio sobre la evolución y desarrollo de los modelos de componente actuales, creados por organizaciones de renombre internacional como: Oracle, OMG y Microsoft. El método **analítico** permite separar en partes el campo de acción —los modelos de componentes—, para así captar peculiaridades y relaciones comunes. A través de la **modelación** se forman abstracciones de los elementos arquitectónicos que permitirán estandarizar los componentes.

Métodos empíricos: para reconocer los problemas de reutilización que identifican los desarrolladores que utilizan el marco de trabajo Sauxe se recurre al método científico de investigación: la **encuesta**. Para validar los resultados de la investigación se hace uso de la **medición**, con el objetivo de obtener información numérica acerca de la reutilización. La **entrevista** fue el método para identificar la muestra escogida para la investigación.

Como ***resultado de esta investigación*** se obtendrá un modelo de componentes para el marco de trabajo Sauxe.

CAPÍTULO 1: FUNDAMENTACIÓN TEÓRICA.

El presente capítulo constituye el marco teórico de la investigación. En el mismo se definen una serie de conceptos necesarios para entender el objetivo fundamental de la misma, profundizando en temas como el desarrollo de software basado en componentes, los modelos de componentes más representativos y los mecanismos de integración de diversos marcos de trabajo.

1.1. Desarrollo de Software Basado en Componentes (DSBC)

El DSBC es el proceso de definir, implementar e integrar o componer en sistemas, componentes independientes débilmente acoplados. Surge a finales de los años 90 como un enfoque basado en la reutilización para el desarrollo de sistemas software. Su creación fue motivada por la frustración de los diseñadores de que el desarrollo orientado a objetos no había conducido a una reutilización extensiva (Sommerville, 2005).

El aumento de la complejidad y el tamaño de los sistemas que demandan los usuarios así como la confiablidad y el limitado tiempo para desarrollar, son aspectos que influyen en la decisión de reutilizar componentes de software en vez de implementarlos.

Este enfoque está tomando fuerza en la ingeniería de software, ya que está asentado sobre algunos de los principios de diseño sólidos que soportan la construcción de software comprensible y mantenible. Los componentes son independientes para que no interfieran en su funcionamiento unos con otros. Los detalles de la implementación se ocultan, por lo que la implementación de los componentes puede cambiarse sin afectar al resto del sistema. Los componentes se comunican a través de interfaces bien definidas, de forma que si esas interfaces se mantienen, un componente puede ser remplazado por otro que proporcione una funcionalidad adicional o mejorada (Sommerville, 2005).

1.2. Componente

Debido a la diversidad de estándares en la industria de la informática y las comunicaciones, el término componente es definido por muchos autores:

De acuerdo con Philippe Krutchen, un componente es una parte no trivial, casi independiente y reemplazable de un sistema que cumple una función dentro del contexto de una arquitectura bien definida. Un componente cumple con un conjunto de interfaces y provee la realización física de ellas (Brown, y otros, 1998).

Szyperski afirma que un componente es una unidad de composición de aplicaciones software, que posee un conjunto de interfaces y un conjunto de requisitos, y que ha de poder ser desarrollado, adquirido, incorporado al sistema y compuesto con otros componentes de forma independiente, en tiempo y espacio (Szyperski, 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).
En consonancia con el Instituto de Ingeniería de Software (SEI, *Software Engineering Institute*), un componente es una implementación opaca de funcionalidad, sujeta a composición por terceros y que cumple con un modelo de componentes (Bachmann, y otros, 2000).
La mayoría coinciden en que los componentes son unidades de composición fundamentales de un sistema y que son independientes. Para esta investigación se utilizará la definición propuesta por Cuncill y Heineman que abarca varios de los elementos comunes de las otras definiciones:
"*Un componente es un elemento software que se ajusta a un modelo de componentes y que puede ser desplegado y compuesto de forma independiente sin modificación según un estándar de composición*" (Councill, y otros, 2001).
En correspondencia con la definición anterior, Ian Sommerville (Sommerville, 2005) afirma que los componentes deben ser:

- Estandarizados: significa que estos deben ajustarse a algún modelo de componentes.
- Independientes: deben poder componerse y desplegarse sin tener que utilizar otros componentes.
- Componibles: las interacciones externas deben ser a través de interfaces definidas públicamente.
- Desplegables: deben ser capaces de funcionar como una entidad autónoma o sobre una plataforma de componentes que implemente el modelo de componentes.
- Documentados: tienen que estar completamente documentados para que los usuarios potenciales puedan decidir si los componentes satisfacen sus necesidades.

1.3. Interfaces

Para permitir que las partes que forman al software puedan interactuar, deben estar declaradas las interfaces de forma clara en cada uno de los componentes. Estas son abstracciones del comportamiento de los componentes, mediante las cuales se realizan todas las interacciones. Una interfaz es un conjunto de operaciones con parámetros, a la que un componente pudiera dar soporte si implementa cada una de estas operaciones. Es la parte del componente que se expone públicamente y de esta forma se oculta sus detalles de implementación, esta característica permite que un componente sea reemplazado por otro que implemente la misma interfaz (Montilva, y otros, 2005). Para que no existan conflictos en el reemplazo, a partir del momento en que se definan las interfaces de los componentes estas deben mantenerse inalterables.

Generalmente los componentes presentan dos tipos de interfaces, las que brindan servicios y las que solicitan servicios de otros componentes, esta última no compromete la independencia porque no demanda el uso de un componente específico que le proporcione dichos servicios. Algunos modelos de componentes definen otros tipos de interfaces como: atributos, propiedades y controladores de eventos.

Las operaciones de los componentes que se expresan en una interfaz, son conocidas como Propiedades Funcionales. La forma de especificar estas propiedades de manera tal que exista un acuerdo entre el proveedor y el cliente del servicio, se denomina: contrato de interfaz. No obstante, en un contrato de interfaz no se pueden expresar otras características del componente concernientes a su desempeño, seguridad, sincronización, calidad del resultado, disponibilidad, latencia, entre otras, conocidas como Propiedades Extra-funcionales (Business object component architecture, 1998).

Beugnard y otros (Beugnard, y otros, 2010) proponen cuatro niveles para la clasificación de las propiedades de un componente:

- Sintáctico: se establecen las características funcionales de los sistemas, que generalmente se especifican en las interfaces.
- Comportamiento: se determinan las condiciones que cumplen las entradas (precondiciones) y las salidas (post-condiciones) de las operaciones. Garantiza que las operaciones no solo sean sintácticamente compatibles sino también semánticamente.
- Sincronización: se asegura que las operaciones se realicen en determinado orden.
- Calidad del Servicio: se expresa cuantitativamente los atributos de calidad de los servicios.

Una forma de especificar un contrato de interfaz es con el uso de lenguajes de definición de interfaces. Aunque este tipo de lenguaje solo llega hasta los dos primeros niveles de clasificación (Propiedades Funcionales), existen otros intentos para que las interfaces expresen las Propiedades Extra-funcionales. Algunos de estos son: el lenguaje de programación Eiffel para propiedades de comportamiento, Object Calculus para propiedades de sincronización y la notación NoFun para propiedades de calidad de servicio (Montilva, y otros, 2005).

1.4. Lenguaje de Definición de Interfaces (IDL, Interface Definition Language)

Un lenguaje para la definición de las interfaces no es un lenguaje de programación sino un lenguaje descriptivo que permite representar las propiedades funcionales de los componentes.

En un IDL también se describe la cantidad y orden de los parámetros que necesitan las operaciones de un componente, así como las estructuras de datos que devuelven estas operaciones. Se puede observar la definición de una interfaz en el lenguaje *Hypertext Preprocessor* (PHP) como equivalente a una descripción IDL.

Una de las ventajas de este tipo de lenguajes es que sirve para crear una visión arquitectónica de los sistemas independiente de cualquier plataforma o tecnología, lo que aumenta las posibilidades de reutilización. El código escrito con el IDL no es útil si no existe un compilador IDL que permite traducir la descripción hecha a un lenguaje específico, generando los módulos, clases o interfaces necesarias para construir los componentes en una plataforma determinada.

1.5. Beneficios e inconvenientes del DSBC

Desde que surge el desarrollo orientado a componentes, los defensores de este enfoque lo ven como una forma rápida y sencilla de crear sistemas por ensamblado de activos reutilizables. Los méritos de este punto de vista son debatibles hoy en día, no obstante, existen beneficios al adherirse a un modelo de componentes.

En primer lugar, promueve la encapsulación y separación de conceptos con el uso de interfaces lo cual aumenta la reutilización del código porque limita las dependencias en la implementación. En segundo lugar se encuentra la capacidad de substitución de los proveedores de componentes. Como las interacciones entre componentes se hacen a través de interfaces bien definidas, es posible crear diferentes implementaciones de un componente y así poder sustituir un proveedor por otro. El uso de este enfoque también permite simplificar las pruebas al software pues los componentes se pueden probar independientemente antes de probar el conjunto completo de componentes ensamblados.

Como en cualquier decisión arquitectural, también existen desventajas en el uso de componentes. En la siguiente tabla se describen las principales dificultades y cómo analizarlas.

Tabla 1 Problemas asociados a la orientación a componentes. Fuente: (Hall, y otros, 2011)

Problema	Descripción	Análisis
Peso	Algunos marcos de trabajos para componentes son relativamente pesados, por lo que no son apropiados para aplicaciones pequeñas.	¿Qué tan complejas son las dependencias de sus aplicaciones? ¿Se necesitan las funcionalidades extras proveídas por el marco de trabajo?

Diagnóstico	Problemas para depurar dependencias ya que se necesitan nuevas herramientas para identificar qué pasa cuando los servicios no son publicados como se debe.	Los problemas de depuración de dependencias se simplifican con herramientas genéricas que se pueden utilizar en la mayoría de los modelos de componentes.
Síndrome del fichero	Problemas en tiempo de ejecución causados por los ficheros de configuración de los componentes, los cuales se tornan difíciles de depurar.	Un Entorno de Desarrollo Integrado (IDE, I*ntegrated Development Environment*) puede ayudar, ya que permite análisis temprano y refactorización de código.

1.6. Modelo de componentes

En un modelo de componentes se definen los parámetros a seguir para la construcción de un componente, su documentación y despliegue, de esta forma se puede asegurar que el conjunto de componentes que conforman el sistema puedan comunicarse e interactuar entre ellos. También describe cómo deben interactuar los componentes así como las capacidades que tienen (ciclo de vida y gestión de la configuración) (Hall, y otros, 2011). El modelo debe especificar las propiedades de los componentes tales como estilos de códigos, estándares de documentación y lenguajes para la definición de las interfaces. De las interfaces también se debe especificar en el modelo, la forma de establecer nombres, parámetros y operaciones. Los elementos básicos que deben tener los modelos de componentes según (Weinreich, y otros, 2001) se resumen en la Figura 1.

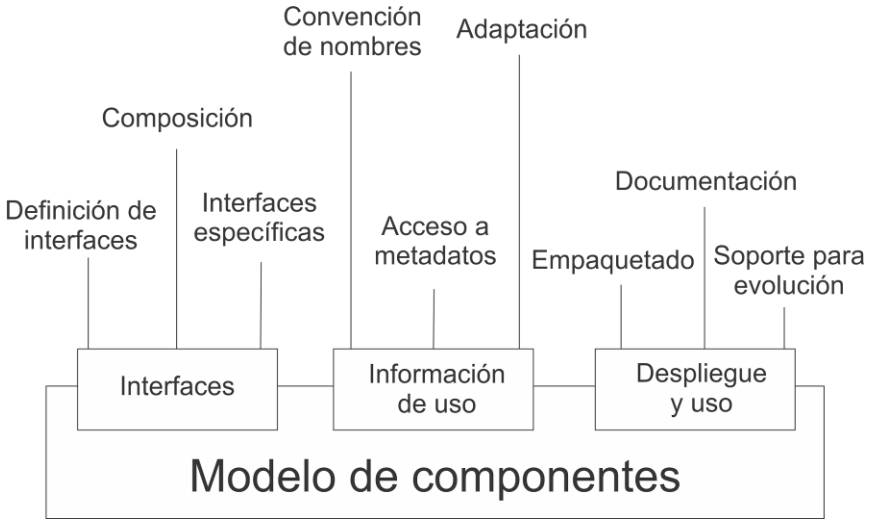

Figura 1 Elementos básicos de un modelo de componentes. Fuente: (Sommerville, 2005).

Los componentes se construyen de forma genérica, por lo que al ser desplegados en un entorno de aplicación particular, el modelo de componentes debe detallar cómo configurarlos para que se adapten al entorno en que serán utilizados. El modelo también debe especificar las reglas para disponer cuándo y cómo se permite el remplazo de componentes, debido al inevitable surgimiento de nuevos requisitos en un sistema.

1.7. Implementación del modelo de componentes

En un ordenador para que los programas puedan ejecutarse deben hacerlo sobre un sistema operativo, el cual brinda un conjunto de servicios genéricos que son utilizados por las aplicaciones. De forma análoga una implementación del modelo de componentes, es el conjunto de elementos que conforman al software necesario para dar soporte a la ejecución de los componentes construidos a partir de un modelo de componentes (Councill, y otros, 2001).

Como forma de disminuir la complejidad y los costos en el desarrollo de los componentes, las implementaciones de los modelos deben incluir servicios que permitan, por ejemplo, la comunicación entre los componentes, la gestión de excepciones y la protección. Usualmente es en los marcos de trabajo donde se soportan estos servicios y se hace cumplir el modelo de componentes.

1.8. Modelos de componentes utilizados en la industria del software

En el mundo se han propuesto varios modelos de componentes, pero si consideramos las valoraciones de (Crnkovik, y otros, 2002) y (Sommerville, 2005) se puede constatar que los más importantes son los siguientes: CORBA de OMG (OMG, 2006), los modelos JavaBeans (Oracle) y Enterprise JavaBeans (Rubinger, y otros, 2010) de Oracle y los modelos .NET (Microsoft) y COM+ (Eddon, y otros, 2000) de Microsoft. En los próximos epígrafes se analizarán –de estos modelos de componentes– los siguientes aspectos: interfaces, implementación, empaquetado y marcos de trabajos donde se ejecutan.

1.8.1. Modelo de componentes JavaBeans

El modelo de componentes JavaBeans creado por Sun Microsystems fue el primer acercamiento, en el lenguaje Java, a la creación de componentes. Es un modelo muy simple de solo 114 páginas de documentación, lo que es un número muy por debajo comparado con los de otros modelos. El alcance de JavaBeans es muy limitado, ya que no es utilizado para crear grandes sistemas distribuidos y solo permite a Java como lenguaje de programación. Esta limitante provoca que el modelo no pueda ser utilizado para la construcción de componentes en el marco de trabajo Sauxe, que utiliza a PHP como lenguaje de programación. El objetivo de exponerlo en este

documento, es conocer su estructura y forma de articular los componentes, experiencia que se puede adquirir para la creación de un modelo propio.

El logro de esta especificación, según Oracle propietario actual de Sun, es que los desarrolladores construyan y vendan componentes Java que se puedan componer en aplicaciones para los usuarios. JavaBeans se diseñó concretamente para la construcción de interfaces gráficas y para que el ensamblado de los componentes se hiciera de manera visual utilizando alguna herramienta. Algo interesante en este modelo, es que los componentes, también conocidos como *beans*, interactúan en dos contextos diferentes: en tiempo de composición, cuando se utilizan dentro de la herramienta de construcción, y en tiempo de ejecución cuando la aplicación que contiene los componentes se está ejecutando.

Para formar la interfaz de un *bean*, el modelo define cuatro elementos: las propiedades, los métodos, las fuentes de evento (*source*) y los oyentes (*listeners*). El uso de los métodos, es el mismo que en un lenguaje de programación, y es la única forma de que el cliente del componente pueda realizar operaciones sobre los demás elementos. Las propiedades son utilizadas para parametrizar el componente en tiempo de composición. Cada propiedad debe tener asociada tres operaciones: obtención de la propiedad, establecimiento de la propiedad, y edición de la propiedad. El uso de eventos es una forma creada en el modelo para la comunicación entre componentes, donde las fuentes generan eventos que son esperados por los oyentes.

La implementación de un *bean*, puede ser un objeto Java o una colección de objetos colaborando entre sí, en los que algunos de sus métodos son los que conforman la interfaz del componente. En la implementación de los *beans* también se permite que objetos externos al componente brinden servicios u operaciones extra-funcionales.

Para ensamblar los componentes no existe en JavaBeans ninguna solución específica. Aun así el modelo de componente fue diseñado para soportar varias formas de ensamblaje, algunas herramientas lo hacen de forma completamente visual, y otras permiten a los usuarios escribir clases que interactúen y controlen a todos los *beans*. Una herramienta que permite la construcción y ensamblado visual de este tipo de componentes, es el IDE NetBeans.

1.8.2. Modelo de componentes Enterprise JavaBeans

Enterprise JavaBeans (EJB) es un modelo de componentes para aplicaciones de negocios distribuidas en el lado del servidor. EJB define un modelo que permite construir un sistema completo a través de la integración de componentes. Cada componente puede representar una colección de procesos de negocios, y estos se ejecutan de manera centralizada en el servidor. La naturaleza distribuida del modelo provee un mecanismo para esparcir componentes a través

de diferentes procesos, máquinas físicas o incluso entre redes **(Rubinger, y otros, 2010)**.

Los componentes EJB intentan complementar la simplicidad de los JavaBeans anteriormente vistos, ya que el nuevo entorno brinda un conjunto de servicios que están especialmente pensados para integrar la lógica de la empresa, de tal forma que el desarrollador no tenga que preocuparse por la programación a nivel de sistema (como control de transacciones, seguridad, computación distribuida, persistencia, etc.), sino que se centre en la representación de entidades y reglas de negocio.

De acuerdo con (Lago, 2005) en su página web, los EJB pueden ser de tres tipos:

- *Beans* de sesión: destinado a ser utilizado por un único cliente. Por ejemplo, para mostrar el cálculo consolidado de todas las inversiones realizadas por el cliente durante la sesión, para mantener un carrito de la compra.
- *Beans* de entidad: son el eje central del modelo de datos, pues se usan para acceder a las bases de datos. El *bean* de entidad gestiona la integridad de los datos persistentes, es decir, la consistencia entre los datos de memoria y los de la base de datos. Por ejemplo: representa a los productos, categorías de productos, cliente, proveedor, cuenta y empleado. Mientras que los *beans* de sesión acaban con la sesión del cliente, los *beans* de entidad duran tanto como los correspondientes datos de la base de datos.
- *Beans* orientados al mensaje: son invocados asíncronamente y utilizan el *Java Message Service (JMS)*. Normalmente un cliente envía un mensaje a la cola del servicio JMS y todos los EJB asociados a la cola destino recibirán el mensaje.

Físicamente un EJB es un conjunto de clases y un archivo XML (*eXtended Markup Language*)[1] que describe el despliegue de dichas clases en un contenedor de EJB. El contenedor EJB es un programa Java que corre en el servidor y que contiene todas las clases y objetos necesarios para la gestión y comunicación de los EJB. Los contenedores pueden beneficiarse del siguiente conjunto de servicios ofrecidos por el servidor:

- Inyección de dependencias
- Concurrencia
- Transacciones

[1] En las últimas versiones de EJB también se puede describir el despliegue de los componentes a través de anotaciones en las implementaciones de las clases.

- Seguridad
- Acceso por nombres
- Interoperabilidad
- Ciclo de vida
- Integración con la plataforma Java Enterprise

Aunque EJB se presenta como una solución de propósito general para simplificar el desarrollo de aplicaciones empresariales, aún no alcanza ese objetivo (Hall, y otros, 2011). Si bien este modelo simplifica muchos aspectos infraestructurales, el desarrollo se complica al tener que definir complejos descriptores de despliegue y código cañería (*plumbing code*) para interconectar la información entre componentes. Es cierto que las últimas versiones (3.*) de la especificación EJB han mejorado en cuanto a simplicidad con el empleo de técnicas de inyección de dependencias y programación orientada a aspectos, aun así muchos desarrolladores consideran que los cambios llegaron tarde. Para el momento en que salieron las nuevas versiones de EJB, ya los desarrolladores se habían establecido en otros marcos de trabajo como OSGi, Spring o Spring Dynamic Modules que también se consideran estándares en la comunidad Java.

Como el modelo EJB es soportado solamente en una plataforma Java, no se puede utilizar para estandarizar la creación de componentes en el marco de trabajo Sauxe, ya que este último utiliza una tecnología diferente. Además, volver a implementar Sauxe en Java es una tarea de muy alta complejidad, que dejaría también sin soporte a los proyectos ya construidos.

1.8.3. Modelo de componentes CORBA (CCM).

CCM es un modelo de componentes del lado del servidor especificado para construir y desplegar aplicaciones basadas en componentes CORBA[2]. Este modelo inspirado en *Enterprise JavaBeans* (EJB) de *Oracle* surgió en junio del año 2002 y fue creado por el *Object Management Group* (OMG) como parte de la especificación de CORBA 3.0. CCM mantiene las características del modelo de objeto de CORBA, lo que permite a los desarrolladores la implementación y configuración de componentes que se puedan integrar utilizando los servicios CORBA, entre los que se incluyen las transacciones, los eventos, la seguridad y la persistencia. Manteniendo los principios abiertos de la OMG, CCM puede escribirse en distintos lenguajes de programación, y hasta permite la interoperabilidad con componentes EJB.

[2] Common Object Request Broker Architecture

La definición de la interfaz de un componente está dada por un conjunto de elementos definidos por la OMG. Las facetas (*facets*), son las interfaces que el componente expone para la interacción con los clientes. Los receptáculos (*receptacles*) que permiten declarar las dependencias de algún agente externo. Las fuentes (*sources*) y sumideros (*sinks*), permiten que los componentes trabajen entre ellos de forma indirecta a través de eventos capturados o emitidos utilizando el patrón de comportamiento *Observer*. Y los atributos (*attribute*) que son valores utilizados para la configuración del componente, que permiten operaciones de acceso, modificación y lanzamiento de excepciones. Para crear o encontrar un tipo particular de componente se utilizan los *homes*, estos manejan el ciclo de vida de los componentes (OMG, 2006).

En la Figura 2 se representan los términos explicados.

Para el ensamblaje, los componentes son conectados al enlazar las facetas (*facets*) con los receptáculos (*receptacles*) y las fuentes de eventos (*sources*) con los sumideros de eventos (*sinks*). Todas las conexiones deben ser explícitamente descritas en un descriptor de ensamblaje, un archivo XML.

La implementación de un componente es un conjunto de segmentos *(*códigos ejecutables escritos en cualquier lenguaje de programación) que implementan al menos uno de los elementos de la interfaz. CCM propone un lenguaje de definición de implementación de componentes (CIDL) para describir los segmentos, el *home* asociado y el tipo de almacenamiento. Diferentes implementaciones se pueden asociar a un mismo componente abstracto.

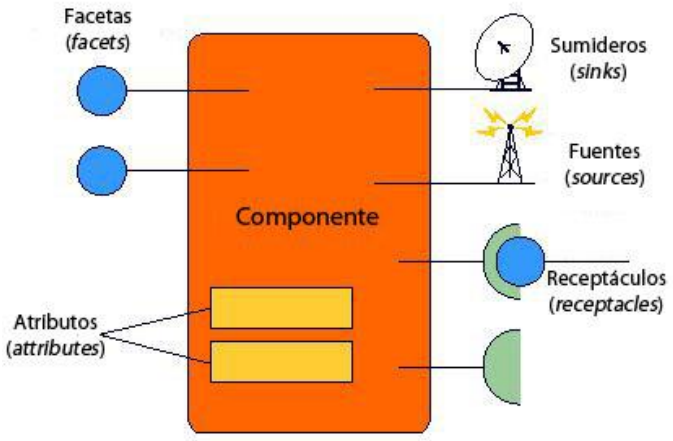

Figura 2 Interfaces de componentes basados en CCM. Fuente: (Bartlett, 2001).

El modelo CCM aprovecha que existen muchos servicios CORBA disponibles, que se pueden utilizar sin cambiar el código fuente de los componentes, permitiendo la reusabilidad y reduciendo la complejidad de los mismos. Para esto CCM utiliza un contenedor, que es un entorno de

ejecución para las instancias de los componentes y sus *homes*. Este contenedor oculta la complejidad de los servicios del sistema y gestiona de manera transparente al componente los aspectos no funcionales como la seguridad, las transacciones y la persistencia. Para hacer esto el contenedor intercepta las llamadas de los componentes, y si es necesario utiliza los servicios solicitados.

A diferencia de EJB el cual está definido junto con un entorno para desarrollar los componentes, CCM es solo el estándar, por lo que necesita de una implementación hecha por alguna entidad externa para que se puedan desarrollar componentes. Hoy en día existen varias implementaciones, algunas de las más conocidas son: TAOX11 (Remedy IT) y ORBIX (Micro Focus).

Pese a que CCM es un poderoso modelo de componentes, es muy complejo. Para poder usar este modelo en la estandarización de los componentes de Sauxe habría que implementarlo en el marco de trabajo además de capacitar a los desarrolladores, lo cual representa un alto gasto de recursos no permisible.

1.8.4. Modelo de componentes COM+

Este modelo pertenece a la compañía Microsoft y está conformado por la unión de varias tecnologías: COM, DCOM y MTS. COM se creó con el fin de proveer heterogeneidad en el uso de los lenguajes de programación similares a C y C++ en la plataforma de Windows, además de crear estándares para interoperabilidad binaria y para las interfaces. DCOM es una extensión de COM, pero que permite la distribución de objetos y MTS deriva de DCOM pero con servicios de transacción y persistencia.

La interfaz de un componente COM es igual a una clase virtual de C++, donde se definen funciones sin asociarle códigos. Para la definición de las interfaces y tipos, se utiliza un IDL llamado Microsoft Interface Definition Language (MIDL). El archivo IDL creado es compilado por el compilador MIDL para utilizarlo en varios lenguajes.

Todas las interfaces que se declaren son derivadas de IUnknown. La interfaz IUnkown consta de tres métodos: AddRef() y Release(), que implementan conteo de referencias y control del ciclo de vida de las interfaces respectivamente; y QueryInterface(), que por especificar un ID permite a una llamada recuperar las referencias a las diferentes interfaces que el componente implementa. La implementación de un componente COM puede ser escrita en cualquier lenguaje de programación en que el compilador siga los estándares de interoperabilidad.

Los componentes COM se empaquetan en ejecutables o librerías dinámicas (DLL). COM soporta dos formas de composición: agregación y contención. En contención, un objeto COM contiene a

otros objetos COM. El objeto externo puede declarar algunas de las interfaces de los objetos internos, y este último las implementa, para que el objeto externo las pueda llamar (delegación), y todo se vea como un solo objeto que implementa todas las interfaces. En la agregación un objeto externo muestra las interfaces de los objetos internos como si esta las implementara pero sin necesidad de hacerlo, la única desventaja de este proceso es que se requiere el código de fuente de los objetos implícitos.

DCOM no cambia el modelo de componentes, solo lo extiende, permitiendo que los componentes que están en distintas estaciones de trabajo se puedan comunicar de forma transparente usando un protocolo de red.

El uso de este modelo para la construcción de componentes en Sauxe sería inadecuado, COM+ está diseñado solo para aplicaciones de *Windows*, lo cual violaría el principio de soberanía tecnológica que persigue el marco de trabajo. También es una limitante el hecho de que este modelo precisa que los componentes sean binarios. El lenguaje con que se desarrolla en el marco de trabajo Sauxe, PHP, no genera binarios porque es un lenguaje interpretado.

1.8.5. Modelo de componentes .Net

.NET es otro modelo de componentes creado por Microsoft completamente diferente de COM+. Se desarrolló con el objetivo de garantizar mayor interoperabilidad de lenguajes y así superar las limitaciones de la interoperabilidad binaria. Para lograr estas características Microsoft mejora la idea de la Máquina Virtual de Java, creando un lenguaje intermedio llamado CIL[3] –similar al Java Byte Code– al cual se traducen todos los lenguajes soportados por .Net. También crea un intérprete para CIL muy similar a la Máquina Virtual de Java.

En .NET la información relativa a las relaciones entre componentes no son especificadas por el usuario en formalismos o ficheros separados, se utiliza un enfoque de lenguaje de programación para desarrollar componentes. Esto significa que los programas contienen esa información en el código y es el compilador en tiempo de ejecución el encargado de relacionar los componentes.

El término para denominar un componente en .NET es Ensamble (*assembly*). Un Ensamble contiene un fichero llamado Manifiesto (*manifest*) donde se describen las siguientes características del componente: metadatos, eventos, recursos y métodos exportados o importados. Este fichero es generado por el compilador el cual produce también el código CIL y lo incluye dentro del Ensamble. Aunque en este modelo no existe un concepto explícito de

[3] Common Intermediate Language

conexión entre componentes, se utiliza una lista de recursos exportados e importados para este objetivo (Crnkovik, y otros, 2002).

Un componente en .NET está conformado por módulos que pueden ser Librerías de Enlace Dinámico (DLL, Dynamic Link Libraries) o ficheros ejecutables. Un módulo por sí solo no puede ser un Ensamble, por lo que no existen jerarquías entre componentes. En la Figura 3 se muestra una representación de la interfaz y la implementación de un Ensamble.

Figura 3 a) Interfaz, b) Implementación de un componente en .NET. Fuente: (Crnkovik, y otros, 2002).

Debido al enfoque de lenguaje de programación de esta tecnología, el marco de trabajo .NET es visto como un elemento fundamental. El marco de trabajo además de contener una librería de clases que maneja la mayoría de las operaciones básicas que se encuentran involucradas en el desarrollo de aplicaciones, también presenta un entorno Común de Ejecución para Lenguajes (*CLR, Common Language Runtime*). El CLR es el encargado de crear componentes con módulos escritos en múltiples lenguajes de programación y lograr que se integren en una misma aplicación.

Si bien .NET soporta un amplio grupo de lenguajes de programación (C#, Visual Basic .NET, Object Pascal, C++, F#, J#, Perl, Python, Fortran, Prolog, Cobol y PowerBuilder), el lenguaje utilizado por el marco de trabajo Sauxe *(PHP)* no es uno ellos. Para lograr que Sauxe se ajuste a este modelo habría que volverlo a implementar usando algunos de los lenguajes soportados, una decisión muy costosa –en tiempo y recursos– para una base tecnológica sobre la cual ya están construidas varios sistemas. Además, el uso de .NET para la creación de proyectos comerciales está sujeto a pagos de licencia, algo complejo de gestionar por problemas del bloqueo económico hacia Cuba.

1.8.6. Comparación entre los modelos de componentes

En la industria del software las grandes compañías han introducido estándares para aumentar la reusabilidad y permitir el desarrollo de componentes compatibles y accesibles. El concepto de componente y las vías para mejorar la reusabilidad varían según los diferentes modelos propuestos. En la Tabla 2 aparece un resumen de algunas características de los modelos de componentes estudiados.

Tabla 2 Modelos de componentes usados en la industria del software.

Modelo de componentes	Empaquetado:	Implementación en:	Plataforma	Interfaces
CORBA	No lo provee	C, C++, Ada, Java	Cualquier plataforma compatible con CORBA	Definidas en IDL
COM+	Binario	C++, Java, Cobol, VB.	Windows	Definidas en MIDL
JavaBeans y EJB	JAR	Java	JVM	Definidas en Java
.NET	Ensamble	C# y los lenguajes soportados por CLR	Plataforma .NET basada en XML	No se definen

Aunque no conviene utilizar directamente ninguno de los modelos estudiados para estandarizar la creación de componentes en el marco de trabajo Sauxe, si es importante tomar características comunes que permitan definir pautas para crear un nuevo modelo de componentes que se ajuste al entorno productivo.

1.9. Marco de trabajo (MT)

Según (Fuentes, y otros, 2001), un MT es el esqueleto de una aplicación que debe ser adaptado a necesidades concretas por el programador de la aplicación.

(Potencier, 2011) afirma que los marcos de trabajo disminuyen 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.

En el entorno que le concierne a esta investigación, se vería entonces un MT como el soporte para crear sistemas a partir de la conformación y reutilización de componentes de software. Como se expuso en epígrafes anteriores, toda arquitectura orientada a componentes necesita de un modelo de componentes, por lo que el MT que encapsule este estilo debe hacer cumplir las especificaciones de un modelo asociado.

El MT debe manejar recursos compartidos por los componentes, y proporcionar mecanismos subyacentes que permiten la comunicación (interacción) entre ellos. Los marcos de trabajo son activos y actúan directamente sobre componentes, por ejemplo administrando su ciclo de vida (activar o desactivar componentes), y otros recursos (Montilva, y otros, 2005).

1.10. Mecanismo de Integración

Todos los marcos de trabajo modernos deben tener un mecanismo de integración que permita comunicar los activos de software que se construyen sobre ellos. Si bien en algunos marcos de trabajo la comunicación es entre componentes que se ajustan a un estándar, existen otros que comunican módulos. Aunque módulos y componentes tienden a parecerse ya que ambos son empaquetados como unidades de despliegue independientes, sirven con propósitos diferentes. Los módulos tratan con empaquetamiento de código y dependencias entre códigos, mientras que los componentes tratan con implementación de funcionalidades de alto nivel y dependencias entre componentes (Hall, y otros, 2011). Para esta parte de la investigación se trataran módulos y componentes indistintamente.

Un concepto importante que aparece en la mayoría de los mecanismos de integración de los marcos de trabajos que se estudiarán, es la **inyección de dependencias**. Esta técnica de implementación permite establecer las dependencias entre componentes o módulos en tiempo de ejecución, esto se hace al referenciar interfaces y no implementaciones directas.

En general, las dependencias son expresadas en términos de interfaces en lugar de clases concretas. Esto permite un rápido reemplazo de las implementaciones dependientes sin modificar el código de fuente de la clase. Lo que propone entonces la inyección de dependencias, es no instanciar las dependencias explícitamente en la clase, sino que declarativamente expresarlas en la definición de la clase. La esencia de la inyección de las dependencias es contar con un componente capaz de obtener instancias validas de las dependencias del objeto e inyectarlas durante la creación o inicialización del objeto. El componente encargado de crear las instancias de las dependencias, realizar las inyecciones y además gestionar sus respectivos ciclos de vida, es denominado **contenedor de inyección de dependencias**. El hecho de que el contenedor a

veces mantenga una referencia a los componentes creados luego de la instanciación es lo que lo hace un contenedor (Guillerón, 2009).

Sin importar si son componentes o módulos debe existir comunicación, por lo que en el presente trabajo se expone cómo fluye la integración en algunos de los marcos de trabajo utilizados en la industria del software. El objetivo del análisis es buscar aportes teóricos y prácticos que puedan ser utilizados en la actualización de Sauxe para que soporte la comunicación entre los componentes y lograr una implementación exitosa de la solución.

Los MT analizados son: Zend Framework 2 y Symfony 2 para PHP además de Spring, OSGi y Spring Dynamic Modules para Java.

1.10.1. Zend Framework 2

Zend Framework 2 (ZF2) es un marco de trabajo de código abierto para desarrollar aplicaciones web y servicios web con PHP 5.3. Es una implementación que usa código cien por ciento orientado a objetos (Zend Technologies, 2012).

Esta nueva versión del marco de trabajo liberada en el 2012, introduce un enfoque al desarrollo de aplicaciones basadas en módulos, que mejora la flexibilidad y organización del código con respecto a versiones anteriores. ZF2 utiliza una nueva arquitectura basada en servicios que se vale de un contenedor de inyección de dependencias para crear los objetos que son solicitados por otros módulos.

Cualquier módulo puede actuar como proveedor de servicios. Para hacerlo se implementa el método getServiceConfig en donde se definen a través de arreglos de PHP los servicios a brindar.

```
class Module {
    public function getServiceConfig() {
        return array(
          'service_manager' => array(
             'services' => array(
               'Auth' => new SomeModule\Authentication\AuthenticationService(),
                )),
        );
    }}
```

Figura 4. Ejemplo de módulo que brinda un servicio llamado “Auth”.

Si se quiere consumir el servicio en algún controlador, se hace uso de un localizador de servicios que se encarga de crear el objeto.

```
$serviceManager = $this->getServiceLocator();
$apiServiceResult = $serviceManager->get('Auth');
```

Figura 5. Ejemplo que consume el servicio “Auth” en un controlador.

Al configurar un módulo en ZF2 se pueden especificar los servicios que se brindan y las dependencias que tienen los servicios, no así las dependencias que tienen los módulos. Esto hace complejo el proceso de reutilización, ya que al portar un módulo hacia otro sistema donde el servicio del que se depende posee otro nombre, se debe inspeccionar el código para encontrar la dependencia problemática y modificarla.

1.10.2. Symfony 2

Symfony es un marco de trabajo de desarrollo web, implementado en lenguaje PHP, que cuenta en la actualidad con dos ramas estables. La versión 2.3 lanzada en mayo 2013 con soporte para tres años y la versión 2.4.1 lanzada oficialmente en enero del 2014.

La base de la nueva filosofía de trabajo de Symfony 2 son los *bundles*: directorios que contienen todo tipo de archivos dentro una estructura jerarquizada. Los *bundles* de las aplicaciones suelen contener clases PHP y archivos web (JavaScript, CSS e imágenes). No obstante, no existe ninguna restricción sobre lo que se puede incluir dentro de un *bundle* (Eguiluz, 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 configuración de los *bundles* se realiza a través del archivo services.yml que se encuentra en la carpeta config/ dentro del *bundle*; en esta configuración solo se especifican 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” en el sistema. Cada servicio se utiliza en toda una aplicación siempre que se necesite la funcionalidad específica que este proporciona.

Al especificar un servicio en el archivo de configuración también se declaran los parámetros que debe recibir este, así como la ubicación de la clase que lo implementa.

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

Figura 6. Ejemplo de configuración de un servicio llamado “Example1”.

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

Figura 7. Ejemplo de consumo a un servicio llamado “example1”.

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.

Una deficiencia notable en Symfony 2, es que al igual que ZF2 sus componentes no declaran sus dependencias, siendo necesario para los desarrolladores inspeccionar el código para la reutilización en otros sistemas.

1.10.3. Spring

Spring es un marco de trabajo 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 del mismo se inició en el año 2003 (Breidenbach, y otros, 2005).

La clase BeanFactory es el contenedor de Spring, cuya responsabilidad es crear y brindar componentes. 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.

En Spring la configuración de un componente se puede hacer a través de archivos XML, anotaciones Java o texto plano. El siguiente código XML muestra cómo se inyecta la dependencia “notifier” dentro del componente “businessService”.

```
<beans>
    <bean id="notifier" class="foo.bar.notify.JmsNotifier" />
    <bean id="businessService" class="for.bar.service.impl.BusinessServiceImpl">
        <property name="notifier" ref="notifier" />
    </bean>
</beans>
```

Figura 8. Ejemplo de configuración de componentes con el contenedor de 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 otros, 2005).

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

1.10.4. OSGi

OSGi es un marco de trabajo 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 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 Máquina Virtual de Java.

Los componentes en OSGi también llamados *bundles* son unidades físicas de modularidad en forma de ficheros JAR que contienen códigos, recursos y metadatos (Hall, y otros, 2011). El fichero META-INF/MANIFEST.MF dentro del JAR es quien contiene las características de modularidad del *bundle*, que pueden ser: nombre, versión, descripción, paquetes externos de los que depende, paquetes internos que exporta y otras. Con esta información el marco de trabajo puede de manera automática identificar *bundles* dentro de la aplicación y ponerlos a disposición de otros *bundles*.

```
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
```

Figura 9. Ejemplo de configuración a través del fichero MANIFEST.MF. Fuente: (Hall, y otros, 2011).

Para brindar servicios, 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.

Al tener definido las interfaces e implementaciones del servicio que se quiere brindar, el próximo paso es registrarlo en el registro de servicios de OSGi utilizando el método registerService del objeto bundleContext.

```
ServiceRegistration registration =
    bundleContext.registerService(interfaces, new LSE(), metadata);
```

Figura 10. Ejemplo de registro de un servicio en OSGi. Fuente: (Hall, y otros, 2011).

Para consumir el servicio desde otro *bundle* se utiliza el método *getService* del objeto *bundleContext*. Este método necesita como parámetro una referencia que le permita buscar sobre los metadatos de los servicios registrados. El resultado devuelto es una instancia de la clase registrada en el registro de servicios.

```
StockListing listing =
    (StockListing) bundleContext.getService(reference);
```

Figura 11. Ejemplo de consumo de un servicio en OSGi. Fuente: (Hall, y otros, 2011).

Este marco de trabajo 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 y el contrato de los servicios se establece a través de interfaces de java.

1.10.5. Spring Dynamic Modules (Spring DM)

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, y otros, 2011).

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, pues estas tareas serán realizadas por el marco de trabajo haciendo uso de las configuraciones en XML.

En la siguiente imagen se aprecia una configuración XML que muestra cómo exponer un *bean* de Spring como un servicio de OSGi. Al hacer uso del *bean* por algún módulo del sistema, Spring DM registra el servicio de manera automática y lo destruye al terminar su uso.

```
<beans>
    <osgi:service ref="myService"
                  interface="com.manning.spring.osgi.simple.MyService"/>
    <bean id="myService"
          class="com.manning.spring.osgi.simple.impl.MyServiceImpl"/>
</beans>
```

Figura 12. Ejemplo de exposición de un *bean* de Spring como un servicio OSGi.

Con este marco de trabajo 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 inyección de dependencias y la transparencia para el cliente de los servicios, respecto al mecanismo de integración.

1.11. Herramientas y tecnologías

Para la implementación del modelo de componentes en el marco de trabajo Sauxe se utilizarán algunas de las herramientas y tecnologías propuestas por el Departamento de Tecnología del CEIGE (Bauta, 2011).

1.11.1. Apache 2.2.22

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, 2013).

1.11.2. NetBeans 7.3

NetBeans es un IDE (por las siglas en inglés de Entorno de Desarrollo Integrado) libre y de código abierto, que con sus 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, 2013). Una de las características fundamentales de NetBeans 7.3 es su editor de código PHP el cual incluye completamiento de código, documentación de las funciones e integración con herramientas de control de versiones.

1.11.3. Visual Paradigm 8

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, 2013).

1.11.4. XML (por sus siglas en inglés de “lenguaje de marcas extensibles”)

Es un metalenguaje extensible de etiquetas desarrollado por el W3C que no se considera como un lenguaje en particular, sino una manera de definir lenguajes para diferentes necesidades. Su aplicación no está solamente enmarcada en Internet a pesar de ser este el medio donde más se utiliza. También puede ser utilizado en bases de datos, editores de texto, hojas de cálculo, entre

otras aplicaciones. Sus creadores lo proponen como un estándar para el intercambio de información estructurada entre diferentes plataformas (García, y otros, 2005).

1.11.5. Lenguaje de programación PHP

Preprocesador de Hipertexto o PHP (por sus siglas en inglés) es un lenguaje de código abierto muy popular especialmente adecuado para desarrollo web y que puede ser incrustado en HTML. Lo que distingue a PHP es que el código es ejecutado en el servidor, generando un HTML que envía al cliente. El cliente recibirá los resultados de ejecutar el script, sin ninguna posibilidad de determinar qué código ha producido el resultado recibido. La ventaja de usar PHP es que resulta extremadamente simple para el principiante, pero a su vez, ofrece muchas características avanzadas para los programadores profesionales (Achour, y otros, 2012).

Quizás una de sus mayores desventajas radica en que promueve la creación de código desordenado, por lo que lo hace muy complejo de mantener.

1.12. Conclusiones parciales

Con el estudio de las particularidades del DSBC, las características de los modelos de componentes más utilizados en la industria del software y los mecanismos de integración de diversos marcos de trabajo se exponen las conclusiones siguientes:

- La estandarización de los componentes del marco de trabajo Sauxe a través de un modelo de componentes, puede permitir la construcción de aplicaciones confiables con posibilidades de reutilización.
- A partir de la revisión de los modelos de componentes más utilizados en la industria del software se identificaron las características comunes siguientes:
    - Definición de componente
    - Definición de la interfaz
    - Empaquetado
    - Ciclo de vida
- Los modelos de componentes estudiados no pueden ser utilizados ya sea por la plataforma que utilizan o por su complejidad de implementación.
- La caracterización de los distintos mecanismos de integración demuestra que la inyección de dependencias es la técnica por excelencia para manejar la modularidad y las dependencias entre componentes.

Por lo antes expuesto se propone el diseño de un modelo de componente para Sauxe que especifique los siguientes puntos: definición de componente, definición de interfaz, meta-modelo

de componente, lenguaje de definición de interfaces, documentación, convención de nombres, versionado, ciclo de vida y empaquetado. Con estos estándares se pretende que los componentes que se desarrollen aumenten sus índices de reutilización.

CAPÍTULO 2: MODELO DE COMPONENTES.

La estructura de este capítulo exhibe la descripción de los elementos del modelo de componentes. Con esta propuesta se pretende brindar una solución efectiva, que actúe como estándar para la construcción de componentes de software reutilizables.

2.1. Definición de componente

Un componente es la unidad básica para la construcción de sistemas, el cual contiene la lógica de negocio que responde a un conjunto de requisitos funcionales. Físicamente un componente es una colección de recursos empaquetados, que contiene metadatos para su almacenamiento y localización en un repositorio.

En una arquitectura cliente-servidor los componentes residen en el lado del servidor, por lo que el cliente solo puede interactuar con los mismos de forma indirecta.

2.2. Definición de interfaz

Todas las interacciones entre componentes se harán a través de su interfaz y es la única vía por la que serán accedidos. Cuando se defina la interfaz para una versión específica de un componente, debe permanecer inalterable. La interfaz de un componente será descrita como un conjunto de puertos que son los puntos de interacción (servicios, dependencias, fuentes y observadores).

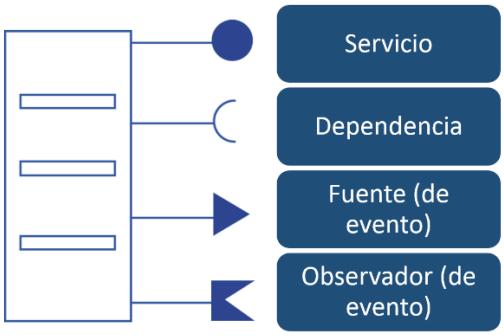

Figura 13 Interfaz de un componente.

- **Servicio**: representa el conjunto de operaciones que brinda el componente para que sean utilizados por otros.
- **Dependencia**: representa el conjunto de operaciones que necesita el componente para funcionar.
- **Fuente**: emite eventos de tipos específicos para uno o más observadores interesados.
- **Observador**: espera que ocurra un evento específico para realizar alguna acción.

2.3. Lenguaje de definición de interfaces para Sauxe (LDIS). Sintaxis y semántica

LDIS es un lenguaje independiente de la plataforma que permite describir las interfaces de los componentes. Con un intérprete para este lenguaje se puede generar un código base para que los desarrolladores completen la implementación.

2.3.1. Convenciones léxicas

La descripción de los símbolos usados en la gramática de LDIS se presenta en la tabla 3.

Tabla 3 Símbolos usados en la gramática.

Símbolo	Significado
→	Está definido como
|	Alternativa
<texto>	No terminal
tk_texto	Tokens (Terminales)
E	Vacío
*	La unidad sintáctica precedente debe repetirse 0 o más veces.
[]	La unidad sintáctica es opcional ponerla

En la Tabla 4 se especifica el significado de cada uno de los *tokens*.

Tabla 4 Especificación de *tokens*.

Token	Lexema	Patrón
tk_System	“System”	System
tk_id	“identificador123”	letra(letra|dígito)*
tk_component	“component”	component
tk_use	“use”	use
tk_provide	“provide”	provide
tk_generate	“generate”	generate
tk_observe	“observe”	observe
tk_attribute	“attribute”	attribute
tk_operation	“operation”	operation
tk_service	“service”	service
tk_event	“event”	event
tk_integer	“integer”	integer
tk_float	“float”	float

tk_string	“string”	string
tk_bool	“bool”	bool
tk_object	“object”	object
tk_void	“void”	void
tk_abre_llave	“{”	{
tk_cierra_llave	“}”	}
tk_dos_puntos	“:”	:
tk_coma	“,”	,
tk_pto_coma	“;”	;
tk_abre_parentesis	“(”	(
tk_cierra_parentesis	“)”	)

2.3.2. Definición de la gramática LDIS

Para formalizar la gramática se utiliza el cuádruplo (N, ∑, P, S) donde:

N: es un alfabeto de no terminales.

∑: es un alfabeto de terminales.

P: es el conjunto de reglas.

S: es el símbolo inicial de la gramática.

La gramática queda entonces definida como:

S:

<inicio>

∑:

tk_System, tk_id, tk_abre_llave, tk_cierra_llave, tk_component, tk_use, tk_dos_puntos, tk_provide, tk_generate, tk_observe, tk_coma, tk_pto_coma, tk_attribute, tk_integer, tk_float, tk_string, tk_bool, tk_object, tk_void, tk_operation, tk_abre_parentesis, tk_cierra_parentesis, tk_service, tk_event

N:

<inicio>, <definiciones>, <masDefiniciones>, <componente>, <atributo>, <servicio>, <evento>, <instComp>, <masInstComp>, decProv>, <decUso>, <decGen>, <decObserv>, <nombres>, <masNombres>, <operacion>, <masOperaciones>, <tipo_dato_op>, <param>, <tipo_dato>, <masParam>, <instEvento>, <masInstEvento>, <operacionEv>

P:

<inicio> → tk_System tk_id tk_abre_llave <definiciones><masDefiniciones> tk_cierra_llave

<definiciones> → <componente> | *<servicio>* | *<evento>*

<masDefiniciones> → <definiciones> <masDefiniciones> | E

<componente> → tk_component tk_id tk_abre_llave <instComp> tk_pto_coma <masInstComp> tk_cierra_llave

<instComp> → <decProv> | <decUso> | <decGen> | <decObserv>

<masInstComp> → <instComp> tk_pto_coma <masInstComp> | E

<decProv> → tk_provide tk_dos_puntos <nombres>

<decUso> → tk_use tk_dos_puntos <nombres>

<decGen> → tk_generate tk_dos_puntos <nombres>

<decObserv> → tk_observe tk_dos_puntos <nombres>

<nombres> → tk_id <masNombres>

<masNombres> → tk_coma tk_id <masNombres> | E

<servicio> → tk_service tk_id tk_abre_llave <operacion> tk_pto_coma <masOperaciones> tk_cierra_llave

<masOperaciones> → <operacion> tk_pto_coma <masOperaciones> | E

<operacion> → tk_operation <tipo_dato_op> tk_id tk_abre_parentesis <param> tk_cierra_parentesis

<param> → <tipo_dato> tk_id <masParam> | E

<masParam> → tk_coma <tipo_dato> tk_id <masParam> | E

<evento> → tk_event tk_id tk_abre_llave <instEvento> <masInstEvento> tk_cierra_llave

<instEvento> → <atributo> | <operacionEv> | E

<masInstEvento> → <instEvento> <masInstEvento> | E

<atributo> → tk_attribute <tipo_dato> tk_id tk_pto_coma

<operacionEv> → tk_operation <tipo_dato_op> tk_id tk_abre_parentesis <param> tk_cierra_parentesis tk_pto_coma

<tipo_dato> → tk_integer | tk_float | tk_string | tk_bool | tk_object

<tipo_dato_op> → <tipo_dato> | tk_void

2.3.3. Declaración del sistema

Todo componente debe ser definido dentro del ámbito de un sistema, el cual es equivalente con el sistema informático real. Por lo tanto, para declarar los componentes y sus interfaces es necesario definir primero un sistema con su identificador:

System *<identificador>* ***{***
...
}

2.3.4. Declaración de componentes

Los componentes se declaran dentro del sistema y se utiliza la siguiente sintaxis:

component *<identificador>* ***{***
*[**provide:** <identificador_servicio_1>**,**<identificador_ servicio_2>**;**]*
*[**use:** <identificador_servicio_3>**,**<identificador_ servicio_4>**;**]*

*[**generate:** <identificador_evento_1>**,**<identificador_evento_2>**;]***
*[**observ:** <identificador_evento_3>**,**<identificador_evento_4>**;]***
}

Un componente debe tener definido al menos uno de los cuatro puertos de su interfaz (*provide, use, generate* y *observ*). En ***provide*** se especifican los identificadores de los servicios que el componente provee. En ***use*** se especifican los identificadores de los servicios brindados por otros componentes los cuales se necesitan para funcionar. Los identificadores que aparecen en un mismo componente en ***provide*** no deben aparecer dentro de ***use***, sería incoherente que un componente necesite algún servicio proveído por él mismo. Un mismo servicio no debe ser brindado por más de un componente.

En ***generate*** se especifican los identificadores de los eventos que el componente genera. En ***observ*** se especifican los identificadores de los eventos generados por otros componentes que se deseen observar para tomar alguna acción si ocurren. Los identificadores definidos en ***generate*** no deben aparecer dentro de ***observ***, debido que un componente no debe esperar sus propios eventos. Un mismo evento no puede ser generado por más de un componente.

2.3.5. Declaración de servicios

Los servicios se declaran dentro de la estructura del sistema utilizando la siguiente sintaxis:

service *<identificador>* ***{***
operation *<tipo_dato_retorno> <nombre_operacion>* ***(*** *[<tipo_dato> <nombre_param1>**,** …]* ***);***
…
}

Los identificadores de los servicios deben ser únicos, y seguirán el patrón determinado en el epígrafe 2.6 *"Convención de nombres"*.

Las operaciones deben especificar siempre el tipo de dato que retorna y un nombre para cada operación. Los nombres de las operaciones deben seguir también el estilo determinado en el epígrafe 2.6 *"Convención de nombres"*. Un servicio debe tener al menos una operación.

Las operaciones también poseen parámetros, estos se especifican definiendo primero el tipo de dato y después el nombre. Cada parámetro es separado por comas, aunque pueden existir operaciones sin parámetros.

Los tipos de datos permitidos son*:* ***int, float, string, bool, object***. En el caso de las operaciones que no retornen valor, se debe utilizar **void**.

2.3.6. Declaración de eventos

Los eventos se declaran dentro de la estructura del sistema y utilizan la siguiente sintaxis:

```
event <identificador> {
        attribute <tipo_dato> <identificador_attributo>;
        …
        operation <tipo_dato_retorno> <nombre_operacion> ( [<tipo_dato> <nombre_param1>,
        …] );
        …
}
```

Las convenciones léxicas para los identificadores y los tipos de datos en los eventos coinciden con la descripción de los servicios.

Los atributos y operaciones del evento describen el objeto que se enviará a los que observen dicho evento. Un evento que no defina atributos y operaciones, no enviará objeto alguno a los observadores, solo los notificará.

2.3.7. Ejemplo de utilización de LDIS

El siguiente ejemplo muestra la creación de un sistema con tres componentes.

```
System Sauxe {
        component Acaxia {
                provide: Seguridad;
                generate: AccesoBaseDatos;
        }
        component Portal {
                use: Seguridad;
                generate: Exception;
        }
        component Traza {
                observ: AccesoBaseDatos, Exception;
        }

        service Seguridad{
                operation int getCertificate(int idUser, int idServer);
                operation int logOut(int idUser, int certificate);
        }
        event Exception {
                attribute int idException;
                attribute string date;
                operation bool saveInTrace();
        }
        event AccesoBaseDatos {}
}
```

En el ejemplo se define un sistema llamado **Sauxe** que contiene tres componentes: **Acaxia**, **Traza** y **Portal**. En el sistema se definen dos eventos: **Exception** y **AccesoBaseDatos**; y también se declara un servicio denominado: **Seguridad**.

El servicio **Seguridad** tiene dos operaciones definidas: ***getCertificate*** y ***logOut***, donde ambas retornan un valor entero y necesitan dos parámetros para funcionar. Este servicio es proveído

por el componente **Acaxia** y es utilizado por el componente **Portal**. En un sistema real el componente **Acaxia** debe encargarse de implementar las operaciones del servicio.

En el evento **Exception** se definen dos atributos: **idException** como entero y **date** como cadena de texto. También se define la operación **saveInTrace** que no necesita parámetros y devuelve un booleano. En un sistema real los atributos y operaciones serán parte de una clase en el componente **Portal** ya que es el encargado de generar este evento. Estos atributos y operaciones se enviarán como un objeto hacia el componente **Traza,** el cual espera la ocurrencia de este evento.

El evento **AccesoBaseDatos** notifica a sus observadores sin enviar objeto alguno. En este caso el evento es generado por el componente **Acaxia** y observado por el componente **Traza**.

2.4. Meta-modelo de componente

Los elementos que se pueden utilizar para modelar un sistema basado en componentes, se resumen en la Figura 14.

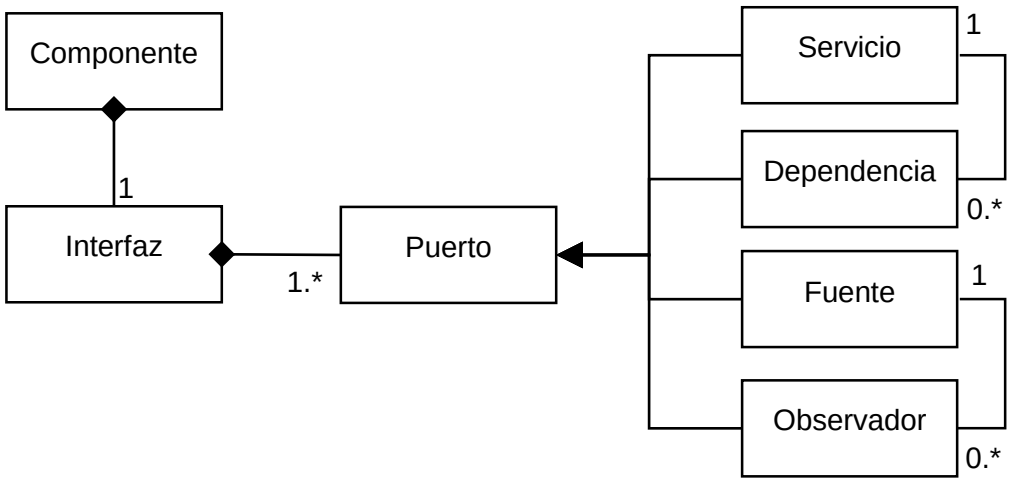

Figura 14 Meta-modelo de componente.

Un componente presenta una interfaz a través de la cual se harán todas las interacciones. Estas interacciones se realizan mediante puertos que pueden ser de tipo: servicio, dependencia, fuente u observador. Un puerto de tipo servicio puede tener suscrito cero o más dependencias. Un puerto de tipo dependencia solo se suscribe a un servicio. De forma análoga ocurre con la relación entre los puertos de tipo fuente y tipo observador.

2.5. Documentación

Proveer una documentación descriptiva de los componentes permite a los clientes comprenderlos y reutilizarlos en aplicaciones para los que no fueron concebidos, siempre y cuando estén construidas bajo el mismo modelo. La documentación de un componente presenta los siguientes datos:

- Nombre del componente
- Descripción general
- Especificación funcional
  - Requisitos funcionales
  - Requisitos no funcionales
- Interfaz
  - Servicios que brinda
  - Servicios de los que depende
  - Eventos generados
  - Eventos observados
- Versión
- Autores
- Contribuidores
- Entidad
- Licencia

2.6. Convención de nombres

La convención de nombres permite que los componentes se puedan distribuir y encontrar sin conflictos. Cada componente utiliza un identificador único que puede contener más de una palabra, todas legibles y que estén relacionadas con su rol en el sistema. Pueden existir varios componentes que presenten el mismo identificador siempre y cuando difieran en su versión. Cada una de las palabras del identificador del componente debe comenzar con mayúscula. Algunos ejemplos de identificadores válidos son: *Acaxia*, *EstructuraComposicion*, *Trazas*, ContabilidadFinanciera.

Los identificadores de cada uno de los puertos son únicos dentro del componente, comienzan con mayúsculas y pueden tener más de una palabra comenzando también con mayúsculas. Algunos ejemplos de identificadores válidos son: *Seguridad*, *ExceptionEvent*, *AccesoBaseDato*.

Los nombres a emplear para las operaciones y atributos se escriben con la primera palabra en minúscula y en caso de que sea un nombre compuesto cada palabra después de la primera

comienza con mayúscula. Con sólo leerlo se reconoce el propósito de la misma, por ejemplo: *getCertificate(), salvarTraza(), idServidor, certificado.*

Cuando se transforme la descripción de las interfaces hecha en LDIS hacia una plataforma específica, los nombres de las clases se formarán con los identificadores de los puertos y sufijos específicos. Para las interfaces que definen las operaciones de los servicios se utiliza el siguiente patrón:

*[IdentificadorComponente][IdentificadorServicio]**Interface***

Por ejemplo: AcaxiaSeguridadInterface

Para las clases de implementación de las operaciones de los servicios se utiliza el siguiente patrón:

*[IdentificadoComponente][IdentificadorServicio]**Impl***

Por ejemplo: AcaxiaSeguridadImpl

Para las clases que implementan las operaciones de las fuentes de eventos se utiliza el siguiente patrón:

*[IdentificadorComponente][IdentificadorEvento]**Event***

Por ejemplo: PortalExceptionEvent

Para las clases que implementan las operaciones de los observadores de eventos se utiliza el siguiente patrón:

*[IdentificadorComponente][IdentificadorEvento]**Observer***

Por ejemplo: TrazaExceptionObserver

2.7. Versionado

Varios componentes con interfaces similares y diferentes versiones pueden coexistir en un mismo sistema, las dependencias serán satisfechas con los componentes cuyas versiones coincidan con las especificadas en su configuración. En caso de no ser especificada se tomará la versión más actual del componente.

La tabla 6 muestra la estructura numérica para conformar los valores de la versión. Cada elemento es separado por el carácter punto (.) y se deben listar todos en el siguiente orden*: (Número principal).(Número menor).(Número micro).*

Tabla 5 Estructura numérica de una versión.

Elemento	Descripción
Número principal	Para actualizaciones mayores donde no se prevé compatibilidad con versiones anteriores.

Número menor	Para actualizaciones funcionales compatibles con las versiones con el mismo número principal.
Número micro	Para reparación de errores.

2.8. Ciclo de vida

El ciclo de vida de los componentes consta de dos fases.

En la primera fase, se “localizan” todos los componentes y se les asigna el estado “no resuelto” a los que tengan alguna dependencia, para los que no tengan dependencias se les asigna el estado “resuelto”. Posteriormente se trata de resolver las dependencias de los componentes no resueltos con los servicios brindados por los resueltos. Finalmente quedan resueltos aquellos componentes que tengan todas sus dependencias resueltas.

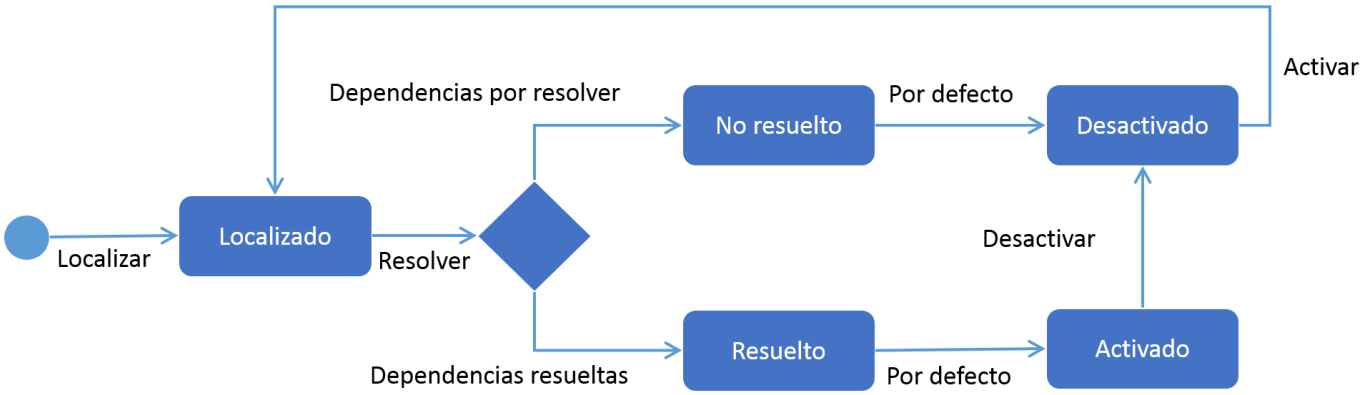

Figura 15 Diagrama de estados: Ciclo de vida de los componentes.

En la segunda fase, los componentes pueden ser “activados” o “desactivados”. Los componentes “no resueltos” están “desactivados” por defecto porque tienen dependencias sin resolver que son necesarias para funcionar. Los componentes “resueltos” son “activados” por defecto, los mismos pueden ser “desactivados” para que no brinden servicios, en ese caso se realiza nuevamente la resolución de dependencias sin contar con ellos. Un componente “desactivado” puede ser “activado” cuando sus dependencias pueden ser resueltas.

2.9. Empaquetado

El patrón de empaquetado cuenta con tres niveles, dos de ellos responsabilidad de la arquitectura de software y un tercer nivel relacionado con las abstracciones propias del proceso de diseño.

1. Nivel Sistema: nivel más alto del encapsulamiento. Agrupa a todos los componentes y al marco de trabajo que implemente el modelo.

2. Nivel Componente: corresponde a la abstracción de los procesos concretos que responden a una solución.
3. Nivel Diseño: corresponde a las abstracciones de diseño (clases, librerías y otros recursos).

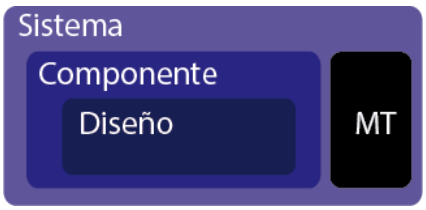

Figura 16 Representación de los niveles de empaquetado.

Un componente puede encapsular a otros componentes que se encuentren en su mismo directorio. Esto no influye en la comunicación ya que todos los componentes de un sistema se encuentran a un mismo nivel de integración. De esta manera todos tienen visibilidad a las interfaces publicadas sin importar su ubicación física en un directorio.

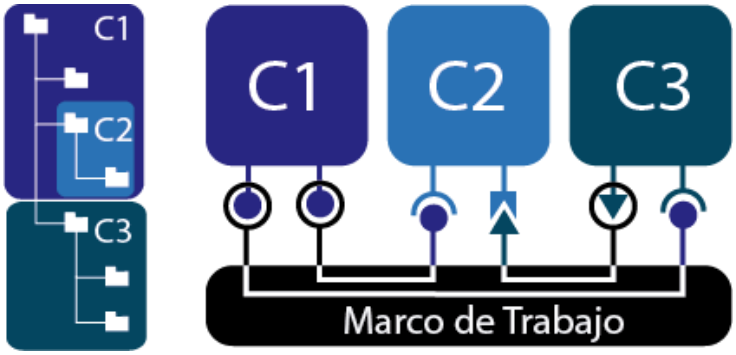

Figura 17 Encapsulación e integración de componentes.

Un componente es considerado una unidad atómica, por lo que todos sus ficheros se encapsulan en un solo archivo, preferiblemente comprimido.

2.10. Conclusiones parciales

Los componentes de un sistema que no se ajusten a un modelo tienen muy poca probabilidad de ser reutilizado, integrado o localizado; precisamente porque no existe un acuerdo que establezca cómo hacerlo. En este capítulo se expusieron los aspectos del modelo que estandarizan los componentes desarrollados sobre el marco de trabajo Sauxe. La definición de componente y la definición de interfaz describen a los diseñadores de componentes qué reglas deben seguir para construirlos e integrarlos. El meta-modelo de componente caracteriza los elementos necesarios para modelar un sistema basado en componentes. El lenguaje de definición de interfaces

independiente de la plataforma permite reutilizar descripciones arquitectónicas de sistemas para desarrollarlos en otros entornos. La documentación, convención de nombres y versionado, son aspectos que inciden positivamente en el mantenimiento y control del sistema. El ciclo de vida especifica que estados deben tener los componentes durante su instalación y despliegue para un mejor control de los mismos. Finalmente el empaquetado brinda información sobre los contenidos que pueden conformar a un componente para que sea reutilizado o ubicado en un repositorio.
Con los elementos especificados previamente queda definido el modelo de componentes para el marco de trabajo Sauxe.

CAPÍTULO 3: VALIDACIÓN DE LA SOLUCIÓN PROPUESTA.

En el presente capítulo se describe la implementación y uso del modelo de componentes en el marco de trabajo Sauxe. De esta manera se valida la aplicabilidad del modelo en un entorno real. Posteriormente se realiza un análisis de los resultados de la métrica propuesta para medir el factor de reutilización en dos componentes de la población declarada. El propósito es demostrar un aumento en la reutilización de los componentes, aspecto definido como variable dependiente de la investigación.

3.1. Implementación del modelo de componentes

Se implementó el modelo de componentes en el marco de trabajo Sauxe con el propósito de incorporar un mecanismo de integración que garantice el desarrollo de componentes reutilizables. A partir de las reglas establecidas en el modelo de componentes y el análisis de los mecanismos de integración de los marcos de trabajos analizados en el capítulo 1, se definieron las características de la implementación.

3.1.1. Librerías para la gestión de los componentes

El marco de trabajo Sauxe presenta en el directorio **/lib/ZendExt** todas las librerías necesarias para su funcionamiento. Para reconocer los componentes, resolver sus dependencias, e integrarlos se implementaron un conjunto de clases que responden a la técnica de **inyección de dependencias**. En la figura 27 se presenta el diagrama de clases correspondiente a la solución. El conjunto de clases que se especificaron en el directorio **/lib/ZendExt/Component** son las siguientes:

- ***Bundle***: es la clase que permite convertir en objetos cada uno de los componentes que se identifiquen en el sistema.
- ***BundleSeeker***: se encarga de buscar en el directorio los componentes definidos y convertirlos en objetos.
- ***BundleSolver***: tiene como objetivo encontrar las relaciones de dependencias y de eventos entre componentes. Modifica el estado de los componentes a resueltos y no resueltos.
- ***BundleProxy***: durante la interacción de los componentes permite controlar el acceso a los mismos y registrar trazas de integración.
- ***Event***: es una interfaz que deben implementar las clases que serán instanciadas por las fuentes de eventos para ser enviadas a los observadores. De esta manera se garantiza que incluyan un constructor.

- ***Factory***: es el encargado de crear la instancia de la dependencia solicitada. Al ser un *singleton* se puede acceder a su instancia única desde cualquier parte del marco de trabajo. Actúa como contenedor dependencias, debido a que almacena las instancias de los objetos solicitados por otros componentes.
- ***FactoryProxy***: actúa como proxy para obtener la instancia única de Factory en los modelos.
- ***Manager***: inicia el proceso de búsqueda y resolución de componentes. Permite almacenar los componentes identificados en la cache para un acceso rápido.
- ***Observer***: es la interfaz que deben implementar las clases que presenten las instrucciones a ejecutar cuando se notifique un evento.
- ***Service***: es utilizada para almacenar las especificaciones de los servicios que brinda un componente.
- ***ServiceDependent***: es utilizada para almacenar las especificaciones de las dependencias que tiene un componente.
- ***SourceEvent***: permite especificar los detalles de las fuentes de eventos de los componentes.
- ***ObserverEvent***: permite especificar los detalles de los observadores de eventos de los componentes.

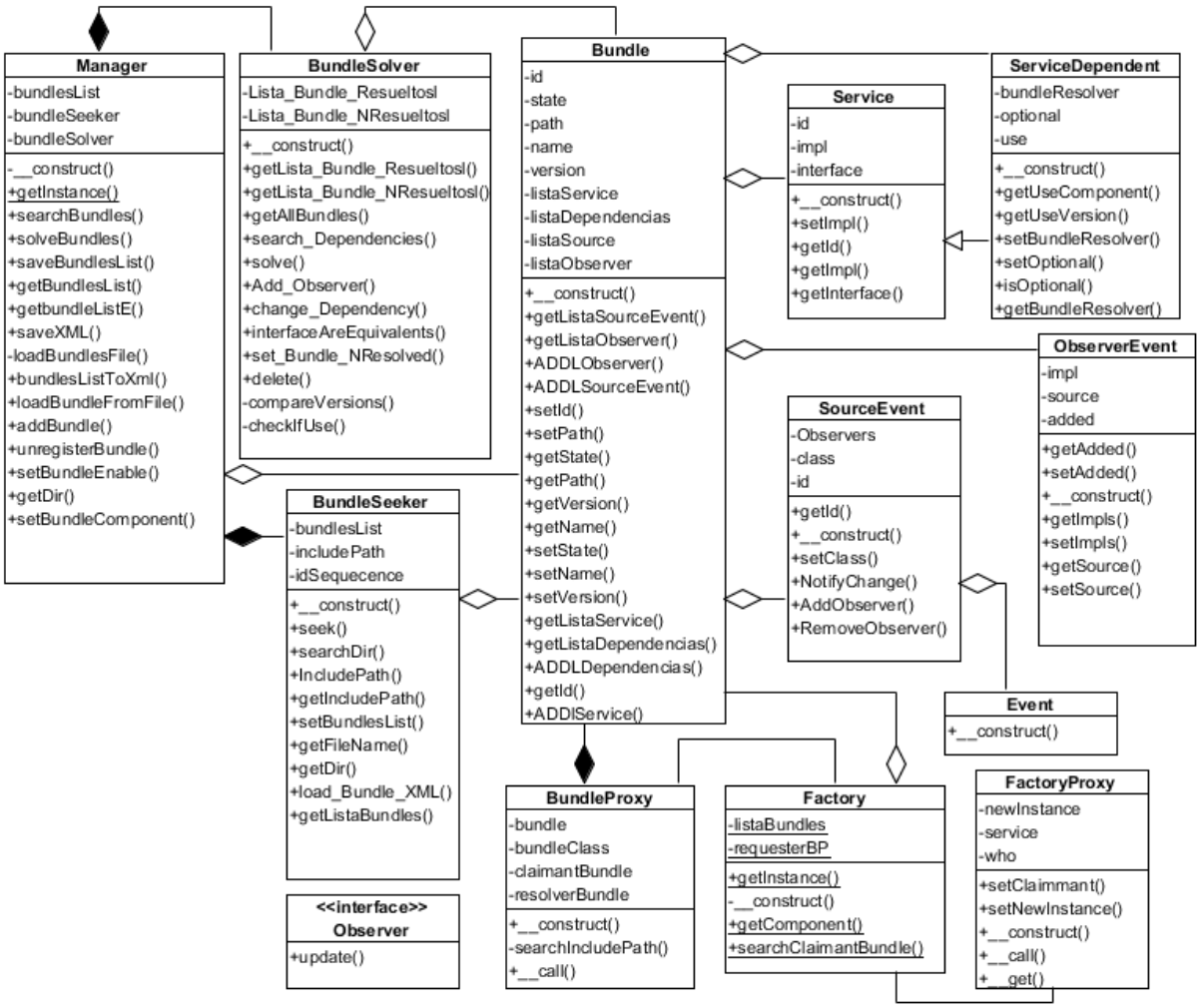

Figura 18 Diagrama de clases de la solución.

3.1.2. Componente

Para el marco de trabajo Sauxe se reconocerá como un componente integrable aquel que en su directorio raíz contenga un fichero de configuración con extensión **.scl**.

Este fichero recogerá la descripción de un componente siguiendo la estructura que se muestra a continuación:

```
<bundle version="" state="">
    <services>
        <service id="" interface="" impl=""/>
    </services>
    <dependencies>
        <dependency id="" interface="" use="" optional=""/>
    </dependencies>
    <sources>
        <source id="" class=""/>
    </sources>
    <observers>
        <observer source="" impl=""/>
    </observers>
</bundle>
```

Figura 19 Configuración de un componente en un fichero .scl.

La etiqueta ***<bundle>*** contiene como atributos la versión (*version*) y el estado (*state*) del componente, elementos necesarios para controlar su ciclo de vida. En el epígrafe 2.7 se explica cómo deben establecerse las versiones. El estado no debe ser especificado por el usuario, el marco de trabajo Sauxe lo asignará al reconocer el componente.

Cada componente contiene una lista de servicios que provee (dentro de la etiqueta ***<services>***) y una lista de dependencias (dentro de la etiqueta ***<dependencies>***), en el caso que se necesiten servicios de otros componentes.

Para declarar un servicio se utiliza la etiqueta ***<service>*** y se especifican los atributos:

- ***id***: identificador para que otros componentes identifiquen el servicio.
- ***interface***: dirección relativa a donde está el fichero PHP con la interfaz donde se definen los métodos que brinda el componente.
- ***impl***: dirección relativa donde está el fichero PHP con la implementación de la interfaz.

Para declarar una dependencia se utiliza la etiqueta ***<dependency>*** y se especifican los atributos:

- ***id***: identificador para que el contenedor de dependencias del marco de trabajo pueda identificar la dependencia.
- ***interface***: dirección relativa donde está el fichero PHP con la interfaz donde se definen los métodos que necesita el componente.
- ***use***: nombre y versión del componente que debe satisfacer la dependencia. Si no se especifica trata de resolver las dependencias con todos los componentes del sistema. El formato es el siguiente: ***nombre-versión*** (ejemplo: “seguridad-1.0.2”).

- ***optional***: determina si es opcional la resolución de la dependencia. Su valor es booleano y por defecto es **false**.

Cada componente contiene una lista de fuentes de eventos (dentro de la etiqueta ***<sources>***) y una lista de observadores de eventos (dentro de la etiqueta ***<observers>***).

Para declarar una fuente de evento se utiliza la etiqueta ***<source>*** y se especifican los atributos:

- ***id***: identificador para que los observadores de eventos puedan suscribirse a fuentes específicas.
- ***class***: dirección relativa a una clase que se instanciará y se enviará a los observadores cuando la fuente los notifique del evento. Este atributo es opcional, debido a que es posible notificar sin enviar información alguna.

Para declarar un observador de evento se utiliza la etiqueta ***<observer>*** y se especifican los atributos:

- ***source***: identificador de la fuente de eventos que será observada.
- ***impl***: dirección relativa a una clase que implementará la acción a ejecutar cuando sea notificado el observador. Esta clase debe contener algún método llamado ***update***, el cual será ejecutado.

Es necesario utilizar las convenciones de nombres especificadas en el epígrafe 2.6 en los identificadores y nombres de clases. De esta manera se evitan conflictos de duplicidad y se garantiza la uniformidad y la organización.

3.1.3. Interfaces

Las interfaces de los componentes serán interfaces PHP, que contendrán anotaciones PHP para especificar los tipos de datos de los parámetros de cada servicio y/o el tipo de dato del retorno, en caso de que existan. A partir de estas anotaciones y los métodos definidos se establecerá la equivalencia entre interfaces. A continuación se presentan las anotaciones necesarias para describir una operación.

```
/**
* Nombre de la operación
* Descripción
* @param tipo_de_dato $nombre_variable - Descripción
* @return tipo_de_dato - Descripción
*/
```

Figura 20 Anotaciones PHP para las operaciones de las interfaces.

Como el contrato entre los componentes se realizará a través de sus interfaces, tanto el componente proveedor como el cliente deberán poseer interfaces equivalentes. Solamente el componente que brinde el servicio implementará la interfaz.

3.1.4. Servicios

Existen dos formas de consumir servicios brindados por los componentes (debe estar declarado previamente en el fichero de configuración como una dependencia). La primera solo puede ser usada dentro de un modelo del sistema.

```
$seguridad = $this->component->Seguridad;
```

Figura 21 Consumo de un servicio desde un modelo en Sauxe.

De esta manera el marco de trabajo devuelve un objeto proxy desde el cual se invocan los métodos definidos en la interfaz del servicio. Este objeto proxy permitirá controlar el acceso del componente al que hace referencia y registrar trazas de integración.

```
$seguridad = $this->component->Seguridad;
$validation = $seguridad->validarCertificado($certificado);
```

Figura 22 Ejecutando una operación del servicio.

También se puede consumir los servicios brindados por los componentes desde cualquier sección de código del marco de trabajo:

```
$factory = ZendExt_Component_Factory::getInstance();
$seguridad = $factory->getComponent($this, "Seguridad", true);
```

Figura 23 Consumo de un servicio desde cualquier sección de código en Sauxe.

El primer parámetro de la operación ***getComponent*** es el objeto que solicita el servicio, el segundo es el identificador de la dependencia y finalmente se especifica con un booleano si se quiere obtener una nueva instancia del objeto proxy. Si ya se solicitó previamente ese servicio se devuelve por defecto la misma instancia del objeto proxy.

Desde los modelos se puede solicitar nuevas instancias del servicio llamando al método *setNewInstace*.

```
$this->component->setNewInstance(true);
$seguridad = $this->component->Seguridad;
```

Figura 24 Obtención de nuevas instancias del servicio desde un modelo en Sauxe.

3.1.5. Eventos

De forma análoga a la utilización de los servicios, para la notificación de eventos existen dos vías.

La primera solo puede ser utilizada desde un modelo del sistema.

```
$this->component->dispatch($eventName, $param1, $param2);
```

Figura 25 Notificación de un evento desde un modelo en Sauxe.

El método ***dispatch*** recibe como primer parámetro el identificador del evento para notificar a todos sus observadores. Si la fuente de evento tiene definida una clase en su configuración, una instancia de la clase se envía a todos los observadores. Los otros parámetros del método ***dispatch*** se utilizarán para la instanciación del objeto.

La segunda vía puede ser utilizada en cualquier parte del marco de trabajo.

```
$factory = ZendExt_Component_Factory::getInstance();
$params = array($param1, $param2);
$factory->dispatchEvent($this, $eventName, $params);
```

Figura 26 Notificación de un evento desde cualquier sección de código en Sauxe.

El primer parámetro de la operación ***dispatchEvent*** es el objeto que notifica el evento, el segundo es el identificador del evento a notificar y finalmente se especifica el conjunto de parámetros necesarios para construir el objeto que se enviará a los observadores.

Si la fuente de evento especifica una clase para que sean enviadas sus instancias a los observadores, esta clase debe heredar de ZendExt_Component_Event.

La clase que implementa la acción a tomar si ocurre un evento observado, debe implementar la interfaz ZendExt_Component_Observer. Este método se ejecuta cuando ocurre el evento esperado.

```
class TrazaExceptionObserver implements ZendExt_Component_Observer {
        function update($obj) {
                //Instrucciones
        }
}
```

Figura 27 Definición de una clase Observer.

3.1.6. Documentación

Para la documentación se utiliza una estructura XML, lo que permitirá la búsqueda y recuperación de componentes en repositorios. El uso de etiquetas XML proporciona a los motores de búsqueda una rápida indexación del contenido, además de garantizar que los documentos estén bien formados a través de un lenguaje de especificación de estructuras.

La descripción se establece en el fichero de configuración del componente (.scl) dentro de las

etiquetas ***<bundle></bundle>.*** El siguiente ejemplo presenta una estructura XML con la documentación de un componente:

```
<doc>
    <name>Acaxia</name>
    <description>
        Acaxia es un sistema que fue diseñado exactamente
        para gestionar la seguridad a cualquier aplicación web que se
        subscriba a él.
    </description>
    <authors>
        <author>Oiner Gomez</author>
        <author>Rene Bauta</author>
    </authors>
    <contributors>
        <contributor>Yuniel Cedeño</contributor>
    </contributors>
    <entity>CEIGE</entity>
    <license>MIT</license>
    <requirements>
        <functionals>
            <req>Gestionar gestor de Base de datos</req>
            <req>Gestionar Servidor</req>
        </functionals>
        <nofunctionals>
            <req>El producto no debe contener palabras en otros idiomas</req>
            <req>
            El sistema solo podrá ser utilizado en territorio cubano
            y por las entidades autorizadas por el Ministerio de las FAR
            </req>
        </nofunctionals>
    </requirements>
</doc>
```

Figura 28 Ejemplo de la documentación de un componente.

3.2. Análisis de resultados del factor de reutilización

Con la finalidad de demostrar una mejora en el factor de reutilización de los componentes desarrollados sobre el marco de trabajo Sauxe, se utiliza la métrica definida en (Rodríguez, 2012) para aplicarla a una muestra. Finalmente se realiza una valoración del impacto de la métrica en la calidad de los componentes aplicados.

3.2.1. Características de la muestra

La población está compuesta por todos los componentes diseñados para el Sistema de Gestión Comercial de Operaciones de Importación y Exportación construido sobre el marco de trabajo Sauxe, para un total de 23. Como el sistema aún está en una fase muy temprana de desarrollo, se realizó una entrevista no estructurada con el arquitecto principal y se le pidió que identificara

los componentes ya avanzados en su construcción, el resultado arrojó un total de 2 componentes que representan el 8,7% de la población. Esta muestra puede ser representativa ya que en concordancia con lo afirmado por Rolando Alfredo (León, y otros, 2011), no está definido un límite mínimo de confiabilidad para poblaciones finitas menores de 30 unidades de estudio.

Los componentes seleccionados son:

- Cartera comercial
- Intercambio

3.2.2. Aplicación de la métrica

A continuación se muestra una relación de las variables necesarias para estimar el valor de la métrica:

Tabla 6 Relación de las variables necesarias para estimar el valor de las métrica.

Variable	Descripción
FA	Factor de alcance del componente.
CR	Capacidad de reutilización del componente.
$\sum MId$	Sumatoria de los métodos de las interfaces del componente que brindan funcionalidad al subdominio de la aplicación, estos métodos se van a usar sin modificación.
$\sum MI$	Sumatoria de todos los métodos de las interfaces del componente (el total de métodos de cada interfaz).
CE	Capacidad de extensión del componente.
$\sum CMI$	Sumatoria de la cohesión de cada método de la interfaz.
$\sum ME$	Sumatoria de los métodos de las interfaces que se van a extender.
FM	Factor de modularidad del componente.
CC	Cohesión del componente.
$\sum EC(MI$	Sumatoria de los elementos del componente que participan en la ejecución de un método (MI).
$\sum EC$	Sumatoria de los elementos por los que está compuesto el componente, que pueden ser clases, librerías y otros.
$\mathbf{P(EC(MI))}$	Peso que se le asigna a la relación que se estable entre los elementos del componente y un método (MI).
$\mathbf{AC}$	Acoplamiento del componente.
$\sum DC$	Sumatoria de las dependencias de otros componentes del dominio.

$\sum C$	Sumatoria de todos los componentes del dominio.
ID	Índice de la documentación de los métodos de la interfaz del componente que se van a usar o a extender.
MD	Método documentado.
$P(MD)$	Peso que se le otorga a la documentación que tiene el método, teniendo en cuenta los argumentos que recibe y los valores de retorno.
CES	Complejidad estática del componente.
R_i	Cantidad de relaciones entre las clases.
$P(R_i)$	Valor de peso de cada relación entre clases.

Teniendo en cuenta la información requerida para las variables de la métrica, se utiliza el diagrama de componentes del Sistema de Gestión Comercial de Operaciones de Importación y Exportación, el cual se muestra en la Figura 29. También se hace necesario para determinar el valor de algunas de las variables de la métrica, contar con los diagramas de clases del diseño que se muestran en la figuras 30 y 31 correspondientes al componente Cartera comercial e Intercambio respectivamente. Todos los diagramas mencionados se diseñaron en el Departamento de Desarrollo de Productos del centro CEIGE.

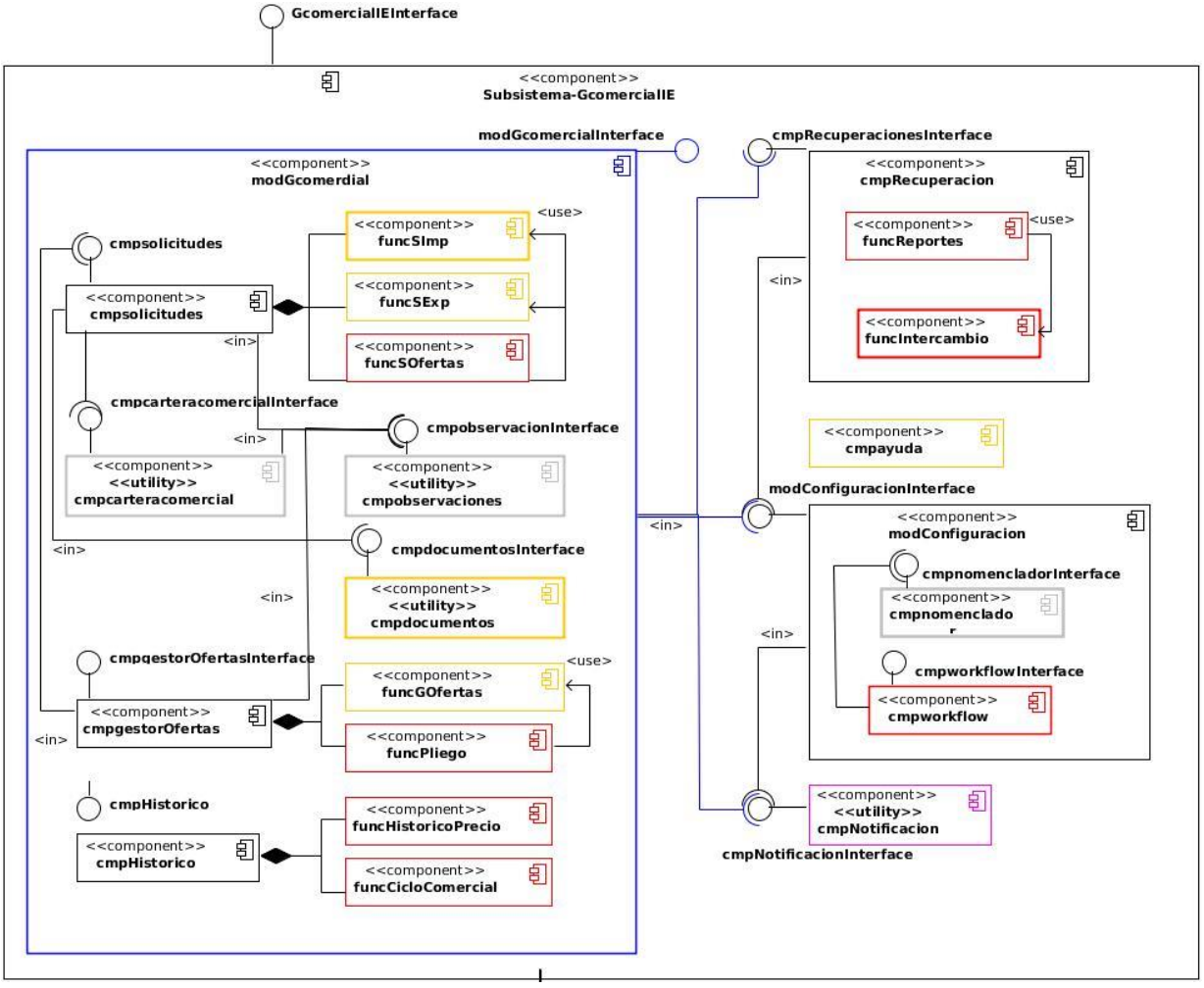

Figura 29 Diagrama de componentes del sistema de gestión comercial de operaciones de importación y exportación.

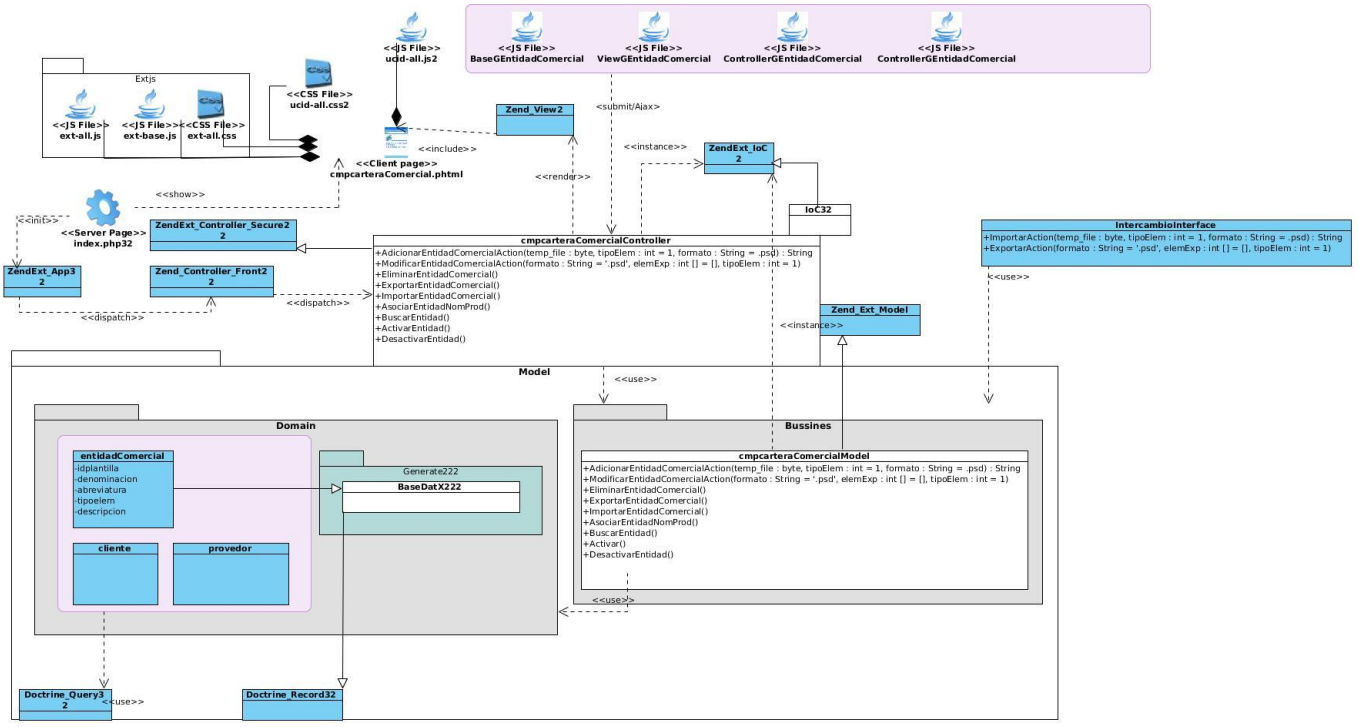

Figura 30 Diseño del componente Cartera comercial.

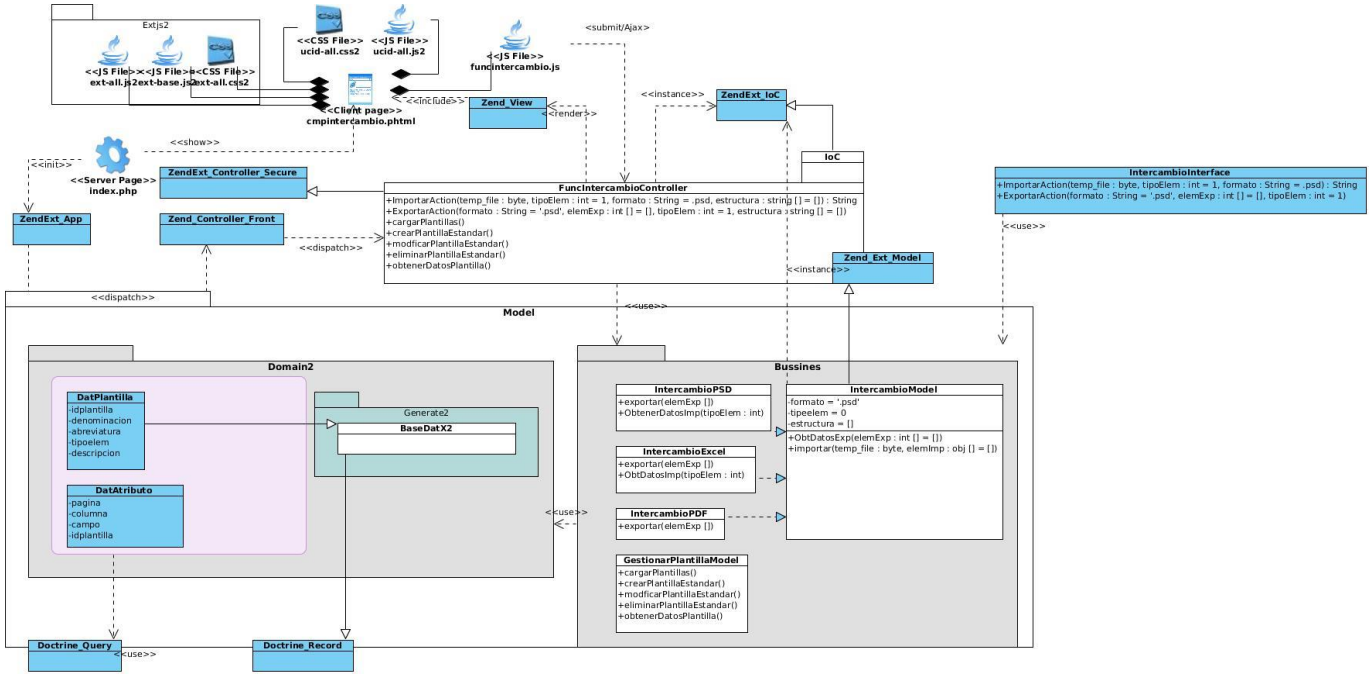

Figura 31 Diseño del componente Intercambio.

3.2.3. Estimación del factor reutilización (FR)

A. La primera variable que se calcula en la métrica es el factor de alcance (FA), que depende de la capacidad de reutilización (CR) y la capacidad de extensión (CE):

A.1. Determinado CR para Cartera comercial:

$$CR = \frac{\sum MId}{\sum MI} = \frac{2}{2} = 1$$

A.2. Determinado CR para Intercambio:

$$CR = \frac{\sum MId}{\sum MI} = \frac{2}{2} = 1$$

A.3. Determinado CE para Cartera comercial:

$$CE = \frac{\sum ME}{\sum MI} = \frac{0}{2} = 0$$

A.4. Determinado CE para Intercambio:

$$CE = \frac{\sum ME}{\sum MI} = \frac{0}{2} = 0$$

A.5. Determinando FA para Cartera comercial:

$$FA = CR + CE = 1 + 0 \ = 1$$

A.6. Determinando FA para Intercambio:

$$FA = CR + CE = 1 + 0 \ = 1$$

Como resultado del cálculo de la primera variable se obtuvo para ambos componentes una **reutilización total**. Esto se debe a que ambos componentes tienen una capacidad de reutilización con valor uno, lo que significa que pueden ser reutilizados en el desarrollo de un dominio de forma total y sin ningún tipo de ajuste o personalización.

B. La segunda variable que se calcula en la métrica es el factor de modularidad (FM), que depende de la cohesión del componente (CC) y el acoplamiento del componente (AC).

B.1. Determinando CMI para cada método de la interfaz de Cartera comercial:

$$CMI_1 = \frac{\sum EC(MI)}{\sum EC} * P\big(EC(MI)\big) = \frac{9}{27} * 0{,}3 = 0{,}1$$

$$CMI_2 = \frac{\sum EC(MI)}{\sum EC} * P\big(EC(MI)\big) = \frac{9}{27} * 0{,}3 = 0{,}1$$

B.2. Determinando CC para Cartera comercial:

$$CC = \frac{\sum CMI}{\sum MI} = \frac{0{,}1 + 0{,}1}{2} = 0{,}1$$

B.3. Determinando CMI para Intercambio:

$$CMI_1 = \frac{\sum EC(MI)}{\sum EC} * P\big(EC(MI)\big) = \frac{14}{27} * 0{,}3 = 0{,}15$$

$$CMI_2 = \frac{\sum EC(MI)}{\sum EC} * P\big(EC(MI)\big) = \frac{14}{27} * 0{,}3 = 0{,}15$$

B.4. Determinando CC para Intercambio:

$$CC = \frac{\sum CMI}{\sum MI} = \frac{0{,}15 + 0{,}15}{2} = 0{,}15$$

B.5. Determinando AC para Cartera comercial

$$AC = \frac{\sum DC}{\sum C} = \frac{1}{23} = 0{,}043$$

B.6. Determinando AC para Intercambio:

$$AC = \frac{\sum DC}{\sum C} = \frac{0}{23} = 0$$

B.7. Determinando FM para Cartera comercial:

$$FM = CC - \mathrm{AC} = 0{,}1 - 0{,}043 = 0{,}057$$

B.8. Determinando FM para Intercambio:

$$FM = CC - \mathrm{AC} = 0{,}15$$

Como resultado del cálculo de la segunda variable se obtuvo para ambos componentes una modularidad **baja**. Esto es provocado por tener los factores de modularidad por debajo de 0,2 lo cual indica que los componentes tienen una baja cohesión y un alto acoplamiento.

Este elemento esta fuera del alcance del modelo de componentes debido a que la cohesión y el acoplamiento dependen del diseño de clases creado para la solución.

C. La tercera variable que se estima en la métrica es el índice de documentación (ID) y se determina sumando los pesos otorgados a la documentación de los métodos de la interfaz.

C.1. Determinando ID para Cartera comercial:

$$ID = \frac{\sum P(MD)}{\sum MI} = \frac{1 + 1}{2} = 1$$

C.2. Determinando ID para Intercambio:

$$ID = \frac{\sum P(MD)}{\sum MI} = \frac{1 + 1}{2} = 1$$

De los resultados obtenidos se infiere que ambos componentes poseen un índice de documentación **alto** debido a que esta variable es mayor que 0,8**.** Este resultado es producto de

que los componentes presentan una documentación que describe cuáles son los argumentos y los atributos que retornan las operaciones del componente, así como las características no funcionales del mismo.

Si se cumple con lo descrito en el modelo propuesto para la documentación de los componentes y sus interfaces, el índice de documentación siempre tiene como resultado: valor uno. Esto garantiza que sin tener en cuenta las demás variables de la métrica, los componentes siempre contarán como mínimo con factor de reutilización adecuado.

Una vez obtenidos todos los elementos de la fórmula para cada uno de los componentes, se determina que:

El factor de reutilización para Cartera comercial es:

$$FR = FA + FM + ID = 1 + 0{,}057 + 1 = 2{,}057$$

El factor de reutilización para Intercambio es:

$$FR = 1 + 0{,}15 + 1 = 2{,}15$$

Según el valor del factor de reutilización de ambos, la métrica indica que los componentes además de cumplir con las características del dominio, son extensibles o personalizables a nuevos entornos sin comprometer sus estructuras. Esto significa que para la toma de decisiones sobre la reutilización de los componentes, se considera que ambos son **recomendables**.

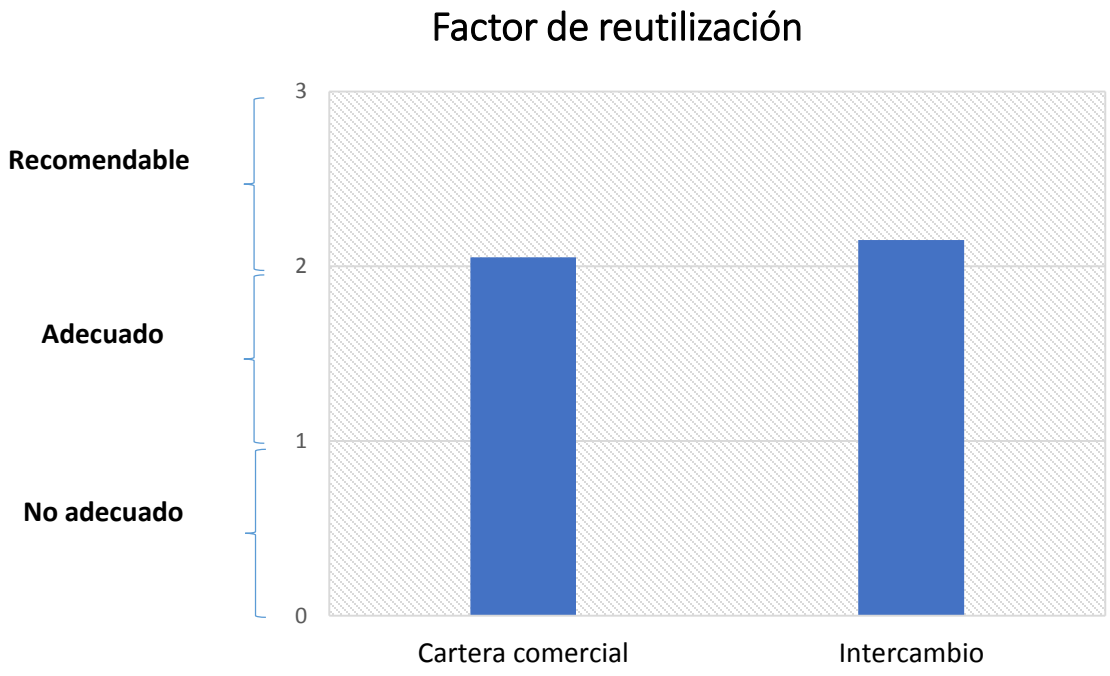

Figura 32 Resultados de la aplicación de la métrica propuesta.

3.3. Conclusiones parciales

La implementación del modelo de componentes establece en términos prácticos, qué es un componente para el marco de trabajo Sauxe y cómo deben interactuar entre ellos. Además, muestra la aplicabilidad del modelo de componentes en un entorno real.

El Sistema de Gestión Comercial de Operaciones de Importación y Exportación es el primero en CEIGE que se comienza a construir usando el nuevo mecanismo implementado. Por esta razón se utilizaron dos de los componentes más avanzados en su desarrollo para medir su factor de reutilización. De esta medición se puede concluir que los componentes Cartera comercial e Intercambio tienen una capacidad de reutilización total, una baja modularidad y un índice de documentación alto. Con estos elementos se puede afirmar que el factor de reutilización de ambos componentes es recomendable. Estos resultados son un indicativo de que el uso del modelo de componentes es una garantía para la reutilización de los componentes construidos.

CONCLUSIONES

La investigación desarrollada en la realización de este trabajo de diploma y el análisis de la documentación estudiada permitió el cumplimiento de los objetivos trazados. Luego del estudio realizado y las razones expuestas en el contenido, se logró establecer un marco teórico para los modelos de componentes en la actualidad. Se concluyó que para construir sistemas basados en componentes es necesario contar con un modelo que estandarice los componentes desarrollados y asegure la reutilización de los mismos en otros entornos para los que no fueron concebidos. Se determinó por analogía que un **modelo de componentes** para el marco de trabajo Sauxe debe especificar: definición de componente, definición de interfaz, meta-modelo de componente, lenguaje de definición de interfaces, documentación, convención de nombres, versionado, ciclo de vida y empaquetado.

El modelo de componentes obtenido permite a los desarrolladores contar con estándares que propicien la construcción, integración, documentación y despliegue de componentes reutilizables. Al ser un modelo tiene un carácter universal, por lo que puede ser implementado en cualquier entorno donde se decida optar por una arquitectura basada en componentes.

La implementación del modelo de componentes en el marco de trabajo Sauxe demuestra que es aplicable en un entorno real. Se seleccionó una muestra de los componentes diseñados para el Sistema de Gestión Comercial de Operaciones de Importación y Exportación para calcular el factor de reutilización. Los resultados arrojaron que los componentes presentan una reutilización recomendable, lo cual es un indicador positivo de que existe un aumento en la reutilización de los componentes estandarizados que se desarrollan sobre el marco de trabajo Sauxe.

RECOMENDACIONES

Se recomienda que el modelo de componentes sea implementado en el marco de trabajo Symfony2 debido a que cuenta con gran popularidad en la Universidad y está respaldado por una gran comunidad. De esta manera se podría crear una arquitectura de referencia para los desarrolladores que utilicen PHP para la construcción de aplicaciones web.

Se propone la construcción de un repositorio de componentes donde los usuarios puedan recuperarlos e integrarlos en sus aplicaciones.

También se exhorta a una extensión del modelo para que soporte la construcción de sistemas con componentes distribuidos haciendo uso de servicios web. Esto contribuiría a la estandarización e integración de sistemas que no compartan las mismas directrices tecnológicas.

BIBLIOGRAFÍA

Zend Technologies. 2012. Guía de Referencia para el Programador de Zend Framework 2. [En línea] 2 de 10 de 2012. [Citado el: 10 de 1 de 2013.] http://manual.zfdes.com/2/es/ref/overview.html.

Achour, Mehdi, y otros. 2012. Manual de PHP. s.l. : PHP Documentation Group , 2012.

Bachmann, F., y otros. 2000. *Technical concepts of component-based software engineering.* Software Engineering Institute. s.l. : Carnegie Mellon University, 2000. Technical report.

Bartlett, Dave. 2001. CORBA Component Model (CCM). [En línea] 1 de 4 de 2001. [Citado el: 8 de 1 de 2012.] www.omg.org/gettingstarted/corbafaq.htm.

Baryolo, Oiner Gomez. 2012. Marco de Trabajo Sauxe. *Ficha Técnica.* s.l. : Brigadas Técnicas Juveniles, 2012.

Bauta, René Rodrigo. 2011. *Arquitectura de software. Vista entorno de desarrollo tecnológico. Proyecto SAUXE.* La Habana : CEIGE, 2011.

Beugnard, Antoine, Jézéquel, Jean-Marc y Plouzeau, Noël. 2010. Contract Aware Components, 10 years after. 2010.

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

Brown, A. W. y C.Wallnau, K. 1998. *The current state of CBSE. IEEE Software.* 1998. págs. 37-46.

Business object component architecture. **Digre, T. 1998.** 5, s.l. : IEEE Software, 1998, Vol. 15.

Calás, Abraham. 2012. *Definición de pautas para la creación de un modelo de componentes para el marco de trabajo Sauxe.* La Habana : Serie CIentífica, 2012. Vol. 5.

Cogoluegnes, Arnaud, Templier, Thierry y Piper, Andy. 2011. *Spring Dynamic Modules In Action.* s.l. : Manning Publications Co., 2011.

Councill, W. T. y Heineman, G. T. 2001. *Definition of a software component and its elements.* Boston : Addison-Wesley, 2001.

Crnkovik, Ivica y Larsson, Magnus. 2002. *Building reliablec omponent-based software systems.* Boston : Artech House, 2002.

Eddon, G. y Eddon, H. 2000. *Inside COM+ Base Services.* Redmon : Microsoft Press, 2000.

Eguiluz, Javier. 2012. *Desarrollo web ágil con Symfony2.* 2012.

Especificación técnica Componente IoC. **Morejón, Yoandry. 2009.** La Habana : s.n., 2009.

Fuentes, Lidia, Troya, José M. y Vallecillo, Antonio. 2001. *Lección 1. Desarrollo de Software Basado en Componentes.* Málaga : Dept. Lenguajes y Ciencias de la Computación. Universidad de Málaga., 2001.

Gamma, Erich, y otros. 2005. *Design Patterns. Elements of Reusable Object-Oriented Software.* s.l. : Addison Wesley, 2005. 0-201-63361-2.

García, I., y otros. 2005. Servicios Web. s.l. : Universidad de Castilla-La Mancha, 2005.

Guillerón, Gastón. 2009. Patrón Inversión de Control (IoC) . *Gln Blog.* [En línea] 12 de 7 de 2009. [Citado el: 12 de 4 de 2013.] http://glnconsultora.com/blog/.

Hall, Richard S., y otros. 2011. *OSGi in action.* Stamford : Manning Publications Co., 2011.

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

Lago, Ramiro. 2005. Enterprise JavaBeans (EJB). [En línea] Enero de 2005. [Citado el: 10 de 12 de 2013.] http://www.proactiva-calidad.com/java/ejb/.

León, Rolando Alfredo Hernández y González, Sayda Coello. 2011. *El proceso de investigación científica.* La Habana : Editorial Universitaria, 2011. 978-959-16-1307-3.

Lucena, Vicente Ferreira de. 2002. *Flexible Web-based Management of Components for Industrial Automation.* s.l. : Universität Stuttgart, 2002.

Mata, Manuel Pérez. 2012. «¿Qué es Symfony?» TecnoRetales. [En línea] 2012. http://www.tecnoretales.com/linux/que-es-symfony/.

McCall, J., Richards, P. y Waters, G. 1977. *Factors in Software Quality.* s.l. : Rome Air Development Center, 1977.

Micro Focus. Orbix® Product Family. [En línea] [Citado el: 4 de 2 de 2014.] http://www.microfocus.com/products/corba/orbix/.

Microsoft. .NET. [En línea] http://www.microsoft.com/net/.

Montilva, Jonás A., Arapé, Nelson y Colmenares, Juan Andrés. 2005. *Desarrollo de Software Basado en Componentes.* Mérida – Venezuela : s.n., 2005.

OMG. 2006. *CORBA Component Model Specification.* 2006.

Oracle Corporation and its affiliates. 2013. NetBeans IDE - Features. [En línea] 2013. http://netbeans.org/features/index.html.

Oracle. Trail: JavaBeans(TM). [En línea] [Citado el: 2 de 2 de 2013.] http://docs.oracle.com/javase/tutorial/javabeans/.

Potencier, Fabien. 2011. *Guía de inicio rápido. Symfony 2.* 2011.

Remedy IT. ZeroC - The Internet Communications Engine (Ice). [En línea] [Citado el: 4 de 2 de 2014.] http://www.remedy.nl/en/taox11.

Rodríguez, Laura Elena Magón. 2012. *Propuesta de métrica para terminar el grado de reutilización de los componentes en Cedrux.* 2012.

Rubinger, Andrew Lee y Burke, Bill. 2010. *Enterprise JavaBeans 3.1.* s.l. : O'Reilly Media, Inc.,

2010. 978-0-596-15802-6.

Sametinger, J. 1997. *Software engineering with reusable components.* s.l. : Springer Verlag, 1997.

Sodhi, J. y Sodhi, P. 1999. *Software reuse: Domain analysis and design process.* s.l. : McGraw-Hill, 1999.

Sommerville, Ian. 2005. *Ingeniería del Software.* Madrid : Pearson Education, S.A., 2005.

Szyperski, C. 1998. *Component Software. Beyond Object-Oriented Programming.* s.l. : Addison-Wesley., 1998.

The Apache Software Foundation. 2013. Apache. *The Apache HTTP Server Project.* [En línea] 2013. http://httpd.apache.org/.

Visual Paradimg. 2013. Visual modeling tool for building enterprise applications. *Visual Paradimg website.* [En línea] 2013. http://www.visual-paradigm.com/product/vpuml/provides/.

Walls, Craig. 2011. *Spring in Action.* New York : Manning Publications Co., 2011. 9781935182351.

Weinreich, R. y Sametinger. 2001. *Component models and component services: concepts and principles. In Component-Based Software Engineering.* Boston : Addison-Wesley, 2001.

ANEXOS

Anexo #1 - **Encuesta para investigación sobre modelo de componentes**

Desde que se vinculó a un proyecto de desarrollo de software, seleccione un aproximado de cuántas veces ha tenido que utilizar códigos de otras personas.

- ☐ Nunca
- ☐ De 1 a 2 veces
- ☐ De 3 a 6 veces
- ☐ Más de 6 veces

¿Cómo usted se desempeñó al utilizar los códigos de otras personas?

- ☐ Muy bien, lo entendí todo
- ☐ Entendí algo, pero necesité ayuda
- ☐ No entendí nada, tuve que empezar desde el principio

Seleccione posibles causas que hayan provocado atrasos en su trabajo:

- ☐ No conozco el lenguaje de programación utilizado
- ☐ No existencia de una documentación de lo que ya existe para poder trabajar mejor
- ☐ No alcanza el tiempo para realizar las tareas
- ☐ Nunca he recibido capacitación sobre el tema en que trabajo
- ☐ No entiendo los códigos de otra persona

¿Sabe usted para qué se utiliza IOC en el marco de trabajo Sauxe?

- ☐ Si
- ☐ No

¿Usted utiliza los estándares de códigos definidos en el centro para desarrollar?

- ☐ Si
- ☐ No

¿Sabe usted si su proyecto utiliza una arquitectura orientada a componentes?

- ☐ Si
- ☐ No

Los modelos de componentes son estándares para la creación, composición y documentación de componentes de software. Seleccione de estos modelos de componentes, los conocidos por usted.

- ☐ CORBA
- ☐ COM+
- ☐ Enterprise JavaBeans
- ☐ JavaBeans
- ☐ Microsoft .NET

GLOSARIO

Componente

Unidad de software independiente y desplegable que se ha definido completamente y a la que se accede a través de un conjunto de interfaces.

CORBA

En computación, **CORBA** (Common Object Request Broker Architecture — arquitectura común de intermediarios en peticiones a objetos), es un estándar que establece una plataforma de desarrollo de sistemas distribuidos facilitando la invocación de métodos remotos bajo un paradigma orientado a objetos.

Desarrollo de software basado en componentes (DSBC)

Desarrollo de software a partir de la composición de componentes independientes y desplegables.

Estandarización

Es la redacción y aprobación de normas que se establecen para garantizar: el acoplamiento de elementos construidos independientemente, la calidad de los elementos fabricados y la seguridad de funcionamiento.

Integración

Acción y efecto de integrar, en este caso componentes.

Interfaz

Especificación de los atributos y operaciones asociados con un componente de software. La interfaz es utilizada como el medio de tener acceso a la funcionalidad del componente.

Marco de trabajo

Estructura genérica en algún dominio específico que puede formar la base de una familia de aplicaciones.

Meta-modelo

Un modelo de modelos.

Modelo de componentes

Conjunto de estándares para la implementación, documentación y utilización de componentes.

OMG

El Object Management Group u OMG (de sus siglas en inglés Grupo de Gestión de Objetos) es un consorcio dedicado al cuidado y el establecimiento de diversos estándares de tecnologías orientadas a objetos, tales como UML, XMI, CORBA. Es una organización sin ánimo de lucro que promueve el uso de tecnología orientada a objetos mediante guías y especificaciones para las mismas.

Reutilización

Proceso de creación de sistemas de software a partir de un software existente, en lugar de tener que rediseñarlo desde el principio.